®

INTEGRATION OF SYSTEMC WITH AN IKOS VIRTUALOGIC EMULATION SYSTEM

A Thesis Presented

by

RAMASWAMY RAMASWAMY

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

September 2001

Department of Electrical and Computer Engineering

INTEGRATION OF SYSTEMC WITH AN IKOS VIRTUALOGIC EMULATION SYSTEM

A Thesis Presented

by

RAMASWAMY RAMASWAMY

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to Professor Russell Tessier for having given me a wonderful research opportunity. I would like to thank him for his encouragement, support and guidance. I would like to thank Ikos Systems Inc. for supporting this work. I would also like to thank Texas Instruments Inc. for providing the Reed Solomon coder design. I thank Professors Ian Harris and Wayne Burleson for their valuable feedback. I would also like to thank my friends for making my stay at Amherst a wonderful one.

ABSTRACT

INTEGRATION OF SYSTEMC WITH AN IKOS VIRTUALOGIC EMULATION SYSTEM

SEPTEMBER 2001

RAMASWAMY RAMASWAMY

B.E., UNIVERSITY OF MADRAS, INDIA

M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by : Professor Russell G. Tessier

With the advent of high complexity system-on-a-chip (SoC) designs, IC verification techniques have taken on an important role in the ASIC design flow. A sizable part of complex SoC verification has to be done at the system level. This poses a unique challenge for logic emulators since internal design wires are not easily accessible. In this thesis, a series of research and engineering tasks are described that integrate SystemC, a new software modeling environment, with an Ikos VirtuaLogic emulation system.

This system allows co-modeling of different portions of an SoC. One portion is specified with SystemC and is verified on a host computer. Additional SoC components are specified in a hardware description language (HDL) and are verified on the logic emulator. As a result of the SystemC/emulator interface, the functionality of an existing Reed-Solomon encoder/decoder is verified. In one experiment, its test-bench is migrated to a SystemC model. In a second experiment, an SoC design using a Reed Solomon coder and a Viterbi decoder was created and interaction between these two components was modeled to demonstrate the usefulness of the interface between SystemC and the emulator in SoC verification.

As SoC complexity increases, traditional methods of verification using HDL software simulators lack the verification horsepower to ensure correct system functionality. Simulation times can be infeasibly, large limiting the use of real-world test cases. Logic emulators along with co-modeling provide a high speed solution to the verification problem. A software model of a testbench sends test vectors through a communication channel to an SoC or its components, implemented on the emulator. Current verification speed is limited by the rate at which test vectors are transferred between the emulator and the testbench through a communication channel.

Another goal of this research is to reduce or completely eliminate this overhead by storing portions of the testbench on the emulator. A significant improvement in verification speed can be obtained since the overhead of sending test vectors through the communication channel is absent.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 Motivation

Enabled by shrinking device sizes, microprocessors, digital signal processors, memory and custom logic are being integrated onto a single chip to form systems-on-a-chip (SoC). Verification of such systems poses unique challenges. The various modules of the SoC must be integrated and sufficient real world scenarios must be run to verify that the system operates as designed. It is beneficial to perform this integration as early as possible in the design flow when the design is still fluid enough to allow for straightforward design changes.

Traditional methods of verification using HDL software simulators have proven to be insufficient in verifying SoCs. These approaches lack the parallelism needed to rapidly verify correct system functionality. With a simulation only approach, software debugging can begin only after first silicon is obtained. If bugs become apparent after fabrication, the designer is forced to undertake a time consuming and costly series of design changes and silicon re-spins.

Logic emulators allow designers to verify at the hardware level before fabrication. An emulator is a reprogrammable hardware system that plugs into a board or system and allows in-system testing during chip design. It is a tool that allows designers to quickly create a prototype of a chip design. It can run at close to real time speeds, is able to ensure correct timing relationships, and can support application software. Logic emulation is also helping to drive the next paradigm in system level design called *concept realization*. With concept realization, designers emulate and verify chips at the highest level of abstraction - the behavioral level, before starting implementation.

It is currently possible to integrate software written in high level languages, such as C/C++, with hardware descriptions written in HDLs, such as Verilog and VHDL by means of remote procedure calls or some form of inter-process communication such as sockets. The overhead of passing data between the two verification domains is often a bottleneck limiting verification performance. In [3], a methodology is presented in which both the software and hardware portions of the system are modeled in C/C++. Despite the performance gains that can be obtained, C/C++ based hardware modeling has yet to be embraced by the EDA industry. Currently, HDLs remain the language of choice to describe hardware systems. Hence, there is a need for a flexible, high speed modeling environment that will allow the designer to integrate hardware models (written in HDLs) and software models(written in C/C++) effectively and to develop real world test cases which will test their interaction.

A hardware/software environment can be obtained by interfacing software systems running on a host workstation with hardware models implemented on logic emulators. This integration enables the designer to link simulators, testbench generation products, software C models and debug tools with the emulator to enhance the verification process. Communication between the software and hardware models is usually performed by means of an interface card on the host workstation to which the emulator is connected. These interfaces typically come with their own application programming interface (API) which provides bi-directional data transfer between the software side and the emulator. The API usually provides functions to control the operation of the emulator. Examples of such interfaces are Q/Bridge [29] from Quickturn for use with Mercury and CoBALT series of emulators and Transaction Interface Portal (TIP) [21] from Ikos Systems for use with the VirtuaLogic and VStation series of emulators.

This capability introduces a new type of verification methodology called *co-modeling*. Traditionally, data exchange between a hardware verification platform and a software model(usually written in C) has either been cycle-based or event-based. Co-modeling involves exchanging data by means of *transactions*. A transaction is a multi-cycle communication sequence between the software model and the design on the emulator. The level of abstraction is raised from cycles and events to transactions. Transactions separate data transfer from clock based timing. Multiple events can be transferred in one transaction. The increase in performance over event and cycle based data transfer is substantial. [5] reports A 320X speedup for emulation over PLI based simulation is reported in [5]. Figure 1.1 shows where co-modeling fits with other current verification technologies.

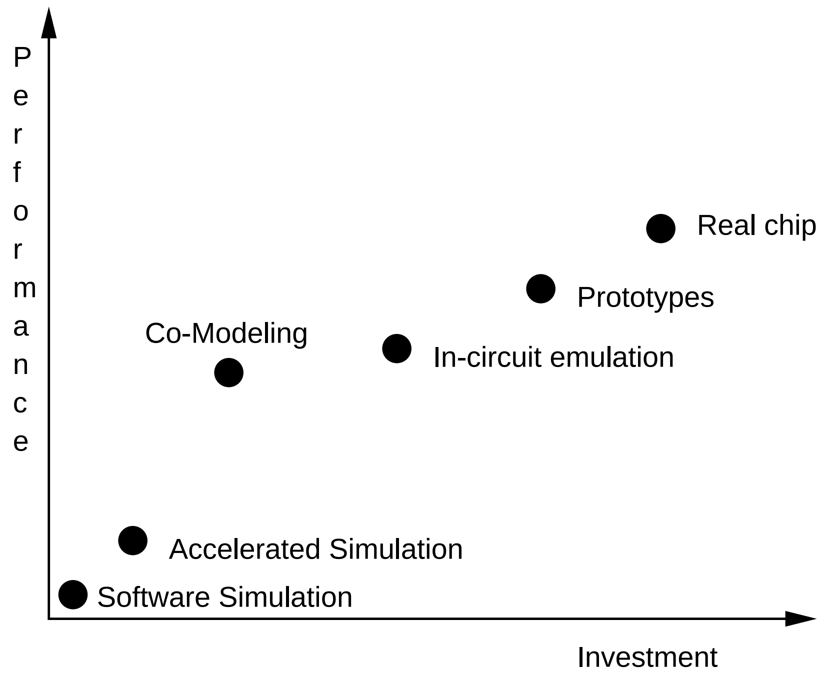


Figure 1.1 Current Verification Technologies

1.2 Objectives

A main objective of this thesis is to increase the scope of applications which use co-modeling. The typical application of co-modeling is verification of a design running on an emulator. The testbench is usually implemented as a C application and test vector transfer with the design under test(DUT) takes place through the co-modeling interface. An alternate verification view point allows a model to simulate the behavior of surrounding hardware and generate emulator input instead of generic test vectors, for the emulator. This type of modeling can be done by using SystemC to describe the software model on the workstation.

SystemC[26] is an extension to C/C++ which consists of a set of class libraries that provide constructs to model hardware components. SystemC provides a simulation kernel that supports cycle-based hardware modeling. As a result, both software and cycle-accurate hardware can be specified in the same language. With SystemC, a designer can create an executable specification of a design at the behavioral level which can be synthesized directly into gates without code conversion to a hardware description language (HDL). This type of modeling enables the designer to accurately simulate hardware-software interactions between various components of a SoC by modeling one component on the emulator and others in SystemC. Testbenches can be modeled at the algorithmic level in addition to test vector representation. Substantial gains in performance can be achieved versus traditional HDL simulation since the cycle based simulation kernel of SystemC is compiled using a standard C compiler. The simulation system allows direct interaction between a portion of a SoC, specified with SystemC and running on a host computer, and additional SoC components undergoing verification on the logic emulator. In our system, the functionality of a Reed Solomon encoder/decoder core has been verified as described in Section 3.4.

Despite the use of transaction based verification and co-modeling, the host workstation-emulator interface is still the verification performance bottleneck. Further performance gains can be achieved for testbench based verification if the testbench is stored on the emulator with the DUT and the verification process is run entirely on the emulator with host intervention required only to start and stop the verification run. We show that an improvement in verification time can be obtained since the the test vector transfer overhead with the co-modeling interface is removed.

1.3 Summary of Chapters

This thesis is divided into 5 chapters. Chapter 2 presents an overview of current verification technologies followed by an introduction to co-modeling and SystemC. Chapter 3 describes the architectural features of the SystemC co-modeling interface and presents the design flows used to verify the functionality of the Reed Solomon coder core. This is followed by a description of an interface implementation for SoC verification. Chapter 4 discusses testbench integration issues and presents the methodology used to migrate the testbench to the emulator. Conclusions and ideas for future work are presented in Chapter 5.

CHAPTER 2

BACKGROUND

2.1 Verification Techniques

2.1.1 Event Based Simulation

The most common type of ASIC verification approach in use today is uni-processor based *event driven* simulation. Changes in design inputs and internal states trigger a chain of signal changes at a particular time which are recorded in a time-ordered event queue. An event driven simulator maintains a list of events to simulate various time instances. As the event list corresponding to the current time is executed, it results in additions to event lists at later points in time. When the simulator completes execution of the current list, the event list for the next time point is executed. Contemporary event driven simulators are between three and four orders of magnitude faster than electrical simulators such as Spice. These simulators typically solve a set of partial differential equations at each time step and trace out analog voltages and currents. Event driven simulators are generally either *interpreted* or *compiled-code*. Interpreted simulators involve little compilation as each event is interpreted at run time. These type of simulators are very fast for small sized designs, but become time consuming for large size designs. Verilog-XL [28] from Cadence is a well known interpreted simulator. Compiled code simulators require a C compiler to convert the HDL source into C code (or some other programming language) and construct an executable file. This type of simulator requires time consuming compilation but achieves a faster runtime, especially for large designs. The Affirma NC Series [30] of simulators from Cadence are examples of compiled code simulators.

2.1.2 Cycle Based Simulation

For large designs event driven simulation can take hours to complete. To improve verification speed, *cycle-based* simulators may be used. These simulators construct event lists only for each clock edge and are limited to functional verification rather than timing. By limiting simulation to clocked edges, the overhead of event management is reduced and an improvement in performance over event driven simulators can be obtained. The Cyclone simulator [31] from Synopsys is an example of a cycle-based simulator.

2.1.3 Hardware Based Verification Tools

For large designs, even a cycle based simulation approach may be overly time consuming. Increased chip complexity and limited design times have resulted in the development of hardware based verification tools such as simulation accelerators and logic emulators. Simulation accelerators typically consist of a number of interconnected processors. A user's design is partitioned across the processors so that logic dependencies are minimized. All the processors operate in parallel offering a significant performance gain over uniprocessor simulation. NSIM [32] from Ikos is an example of such a system.

Logic emulators fully implement user designs in hardware. The emulator provides a complete functional implementation of the design that runs within an order of magnitude of real time speed. One limitation of emulation is that it retains only the functional behavior of the circuit, which means that validation of both performance and timing features cannot be performed on a logic emulator. Two main approaches to logic emulation exist today - the custom processor approach and the field programmable gate array (FPGA) approach. CoBALT [29] from Quickturn uses a custom processor approach. RTL code is partitioned and scheduled to execute on an array of customized concurrent processors. Celaro [33] from Mentor Graphics is another example of an emulation system that uses customized programmable devices.

The inherently reprogrammable nature of FPGAs make them an ideal choice for use in logic emulators. FPGAs are flexible and do not require the fabrication cost of custom processors. Both System Realizer [36] from Quickturn and Virtua-Logic/VStation [27] from Ikos are FPGA based systems. System Realizer contains crossbar interconnection devices to overcome inter-FPGA pin limitations. Ikos emulators use time division multiplexing of the FPGA interconnect (called VirtualWires [4]) instead of crossbars to overcome pin limitations.

Logic emulation shares many of the advantages and disadvantages of both prototyping and software simulation [12]. Like a prototype, the design to be tested is implemented in hardware so that it can achieve high performance during testing.

However, like software, the emulated design can be easily altered and observed to isolate bugs. Transforming a circuit description into a form that can implemented onto an emulator can take many hours to perform. This task is usually done by emulation system software. The high performance of emulators can be attributed to the fact that they can implement the complete circuit in parallel which software simulation and simulaton accelerators cannot do.

For circuits that execute software programs, the emulator can be used to debug these programs much earlier in the design flow than a prototype. This is because an emulator can use a high level description of the circuit (such as a hardware description language) for implementation, while a prototype cannot be made until the complete circuit has been designed. Software simulation is too slow to run software in a feasible time frame. A circuit implemented on an emulator can be inserted into a target environment and the system can be evaluated in a more realistic setting. This helps both to debug the circuit and to test circuit interfaces. For example, an ASIC and the board that will contain it are often developed simultaneously. An emulated version of the ASIC can be inserted into the circuit board prototype for testing both ASIC and board functionality.

2.2 Transaction Based Verification

Synchronization between different verification engines (netlist, RTL, or ISS siumlators and emulators) plays a crucial role in determining the raw performance that can be achieved [5]. Event and cycle based synchronization are examples of fine grained synchronization in which the verification engines synchronize at every event and clock cycle respectively. Due to this tight coupling, the entire system proceeds at the rate of the slowest domain. As a result, verification performance is limited.

An alternative approach is to synchronize the engines only when necessary via *transactions*. A transaction can be defined as a multi-cycle communication sequence between two verifcation domains. Transactions contain both data and synchronization information. A single transaction results in multiple cycles of work being performed by a verification engine. A transaction can be as simple as a memory read or as complex as the transfer of an entire structured packet through a channel. This is accomplished by the transfer of a single message through one synchronization point. The increase in performance over cycle and event based synchronization is substantial.

2.3 Co-Modeling

Co-modeling is a transaction based verification approach in which system behavior (modeled in a high level language such as C), interacts with an RTL design which is implemented on the emulator.

Co-modeling has the following advantages :

- Co-modeling provides a system level verification solution that links system implementation (such as a gate level HDL model) with system behavior (such as a C/C++ based model). It makes use of the fact that the system designer has created a behavioral model of the system and *reuses* this behavioral model as part of the test environment for the entire system

- Co-modeling offers higher performance when compared to an event/cycle interface based on a Programming Language Interface (PLI) implementation
- By using a co-modeling based methodology, we can build interfaces that are standard and re-usable. For example, if an Ethernet transaction model is built, it can be re-used for future designs that might have an Ethernet port built in.

Co-modeling can be used in two ways - *data streaming* and *reactive co-modeling.* This thesis focuses on data streaming. In data streaming, transactions are independent of each other and are sent continuously from the user application to the DUT and vice versa. The data passed between the models is intended to correspond to specific pins of the DUT. A *transactor* located on the emulator does not modify or process the data in any way, but provides the handshaking signals which enable data exchange between the software model and the DUT also located on the emulator. This kind of operation provides maximum throughput. In reactive co-modeling, the transaction sent by the user application depends on the previous transaction. The user application has to wait for the DUT to process the current transaction before it can send a new one. The channel is not utilized effectively in this way since the user application and DUT may be idle awaiting new transactions.

2.3.1 Transaction Interface Portal (TIP)

Transaction Interface Portal [23][24][25] is a transaction based co-modeling interface that provides a communication channel between a host workstation and an Ikos VirtuaLogic/VStation logic emulator.

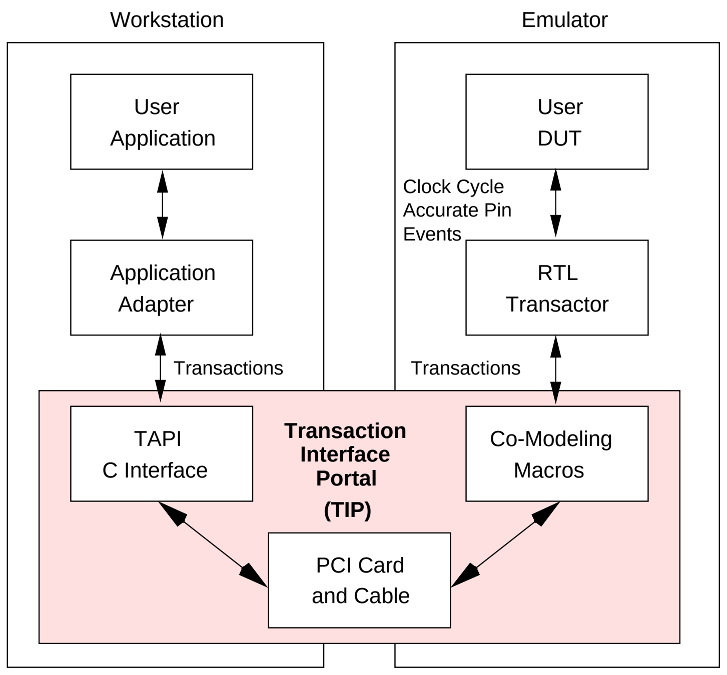


Figure 2.1 Transaction Interface Portal

Figure 2.1 illustrates the layered architecture of TIP for an Ikos emulation system. It consists of a design environment (DE) running on a host workstation, and a DUT running on the emulator. These two domains are connected via a communication channel which provides a mechanism for transporting data and synchronization between the DUT and DE. One main component of the DE is the *User Application*. The User Application may contain a complex C model of a system component, or it may contain a test environment that provides test vectors for the DUT. It utilizes an Application Programming Interface (API) called the Transaction API (TAPI) to communicate with the DUT. This API is provided by the *Application Adapter*. The DUT is comprised of the *User's Netlist*, an *RTL Transactor*, and *Co-Modeling Macros*.

The User Netlist is a gate level model of the design to be verified with the C code. The transactor acts as an interface between the User's Netlist and the underlying co-modeling macros. The co-modeling macros are responsible for data transfer between the transactor and the PCI card. This card is installed on the host workstation and performs data transfer between the emulator and the workstation.

2.3.2 System Operation

Transactions can be initiated by the user application or the DUT. The user application starts a transaction by calling the appropriate API routine with data. This call sends the transaction across the communication channel and activates the co-modeling macros on the emulator. If the DUT's co-modeling primitives are busy, the transaction is buffered in the channel. Once the transaction is received, it is passed to the RTL transactor. The transactor unpacks the data into a sequence of cycle level stimuli which are then applied to the DUT. The reverse process occurs when the DUT initiates the transaction. Similar channel buffering will occur if the co-modeling macros are busy.

The application adapter's API provides a variety of C routines to facilitate sending and receiving of transactions. A summary of the most commonly used TIP functions is shown in Table 2.1. In addition to encoding and decoding transactions, the RTL transactor performs the following functions :

- Negotiate the handshaking signals by which data flow is managed through the co-modeling macros.

- Implement clock control of the DUT so that the DUT does not run when there is no data ready for it.

API call	Function
tapiCOMI1_initInterface()	Initialize communication channel
tapiCOMI1_done()	Close channel, free the card and memory
tapiCOMI1_read()	Read a value from the interface
tapiCOMI1_write()	Write a value to the interface
tapiCOMI1_newtapiDVVForWrite()	Creates a new object for writing
tapiCOMI1_newtapiDVVForRead()	Creates a new object for reading
tapiCOMI1_VLEconnect	Connect to the emulator
tapiCOMI1_VLEconfigure	Download a design onto the emulator
tapiCOMI1_VLEenable	Enable I/O pods on the emulator
tapiCOMI1_VLEdisable	Disable I/O pods on the emulator
tapiCOMI1_VLEspeed	Set the emulator speed index
tapiCOMI1_VLEquit	Terminate emulator connection

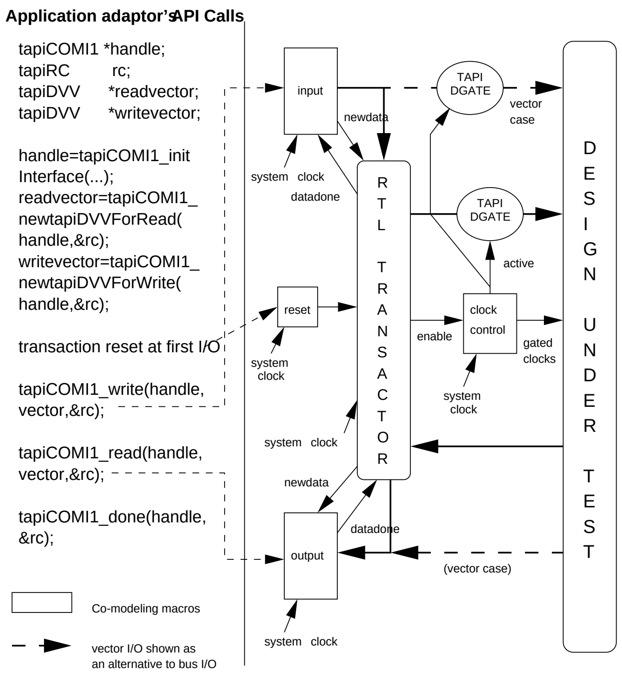
Table 2.1 Common TAPI functions

In addition to encoding and decoding transactions, the RTL transactor performs the following functions :

- Negotiate the handshaking signals by which data flow is managed through the co-modeling macros.

- Implement clock control of the DUT so that the DUT does not run when there is no data ready for it.

The co-modeling macros are a collection of HDL components provided to the user. They perform low level synchronization between the physical channel and the transactors and provide the functionality upon which transactors are built. There are 5 basic primitives as shown in Figure 2.2 - a clock macro, an input macro, an output macro, a dgate macro and a reset macro. The clock macro is a controlled clock generator which can produce a controlled version of any clock in a design clock domain. The clock macro also activates the dgate macro which contains latches that hold DUT data stable at times when the controlled clock is inactive. The dgates isolate the DUT from activity on the outputs of the RTL transactor which could be invalid DUT inputs. The input macro presents data from the communication channel to the user's netlist. The data is sent by the user application through the API call tapiCOMI1_write(). The output macro allows the user's netlist to send data to the user's application. The reset macro is used to initialize the transactor to a known state before the co-modeling session commences.


Application adaptor's API Calls

```
tapiCOMI1 *handle;
tapiRC       rc;
tapiDVV     *readvector;
tapiDVV     *writevector;

handle=tapiCOMI1_init
Interface(...);
readvector=tapiCOMI1_
newtapiDVVForRead(
handle,&rc);
writevector=tapiCOMI1_
newtapiDVVForWrite(
handle,&rc);

transaction reset at first I/O

tapiCOMI1_write(handle,
vector,&rc);

tapiCOMI1_read(handle,
vector,&rc);

tapiCOMI1_done(handle,
&rc);
```

Figure 2.2 TIP Software Architecture

Figure 2.2 illustrates how the co-modeling macros interact with the transactor and the DUT. The input macro allows a netlist running on the emulator to receive data sent by the host using one of the write functions of the TIP API. The macro contains an input data register which temporarily holds channel data until the DUT is ready to access it. Upon the arrival of new data, a signal *newdata* is asserted. Assertion of *datadone* by the transactor indicates that the macro may overwrite the data value during the next cycle. *Newdata* will not be reasserted until *datadone* is deasserted. Similarly, the output macro allows the user's netlist to send data to the user's application through the transactor, where it can be read using the API call, tapiCOMI1_read(). When the macro senses *newdata*, data is read into an output register. Once the read operation is completed, *datadone* is asserted.

2.3.3 Co-Modeling Algorithm

The following steps have to be performed to initiate and close a co-modeling session. The corresponding TAPI function calls are shown along with a short description of their operation.

- Initialization

  1. tapiCOMI1_initInterface()

     This must be the first routine called because it performs two functions - initialize the SPCI card and return a handle to the user's communication interface which all other functions use.

  2. tapiCOMI1_VLEconnect()

Once the SPCI card is initialized, the next step is connection to the emulator. This routine must be called before any other emulator control functions can be used.

3. tapiCOMI1_VLEconfigure()

   Once the SPCI card and the VStation are initialized, the design can be downloaded onto the emulator by calling this function.

4. tapiCOMI1_VLErunInternalClocksDefault()

   This function sets all user clocks in a default manner. It is not appropriate when the design has multiple clocks and precise phase relationships have to be maintained, or when the clock characteristics have to be modified for optimal operation. In these cases, the following functions can be used.

   1. tapiCOMI1_VLEgetClockInfo()

      This function returns minimum low duty cycle and minimum high duty cycle values given a particular clock name. The numbers are given in vcycles which is the base clock period of the emulator.

   2. tapiCOMI1_VLEsetupInternalClock()

      This function allows the user to set up a given clock by name. The clock parameters are specified as arguments to the function. This function can be called multiple times to define multiple clocks in the design.

   3. tapiCOMI1_VLEstartInternalClocks()

This function starts all the clocks specified with tapiCOMI1_VLEsetupInternalClock(). This function must be called prior to calling tapiCOMI1_VLEenable().

5. tapiCOMI1_VLEenable()

   This function enables I/O pods on the emulator.

6. tapiCOMI1_newtapiDVVForWrite()

   This function creates a data structure object for sending data to the emulator. The size of the object is created according to the size specified in the co-modeling macros.

7. tapiCOMI1_newtapiDVVForRead()

   This function creates a data structure object for reading data from the emulator. The size of the object is created according to the size specified in the co-modeling macros.

- Data Transfer

  1. Initialize SystemC model.

  2. tapiCOMI1_write()

     This function sends data that has been written into the object created by tapiCOMI1_newtapiDVVForWrite() to the emulator.

  3. tapiCOMI1_read()

     This function reads data coming from the emulator into the object created by tapiCOMI1_newtapiDVVForRead().

- Closing

    1. tapiCOMI1_VLEdisable()

        This function disables the I/O pods on the emulator and is required to shut down the emulator.

    2. tapiCOMI1_VLEquit()

        This function terminates the connection with the emulator.

    3. tapiCOMI1_done()

        This routine frees up the memory resources allocated to the SPCI card.

2.3.4 Co-Modeling for SoC Verification

Co-modeling with a C application has many limitations when used for SoC designs. The most common use of co-modeling is validation of a design on the emulator with a C testbench. This application involves passing test vectors to and from the design on the emulator. This poses problems for SoC designs in which the interaction between various components of a SoC have to be verified. It is more appropriate for the C model to generate the test vectors in a manner that is similar to a real world situation, i.e. a component driving a hardware model which in this case is emulated.

Two disadvantages exist in the C based approach. Firstly, C by itself cannot be used to model the operation of hardware since it lacks the necessary constructs for concurrency and reactive behavior. Secondly, the use of a functional or behavioral specification written in C may not model the interaction between the SoC components in the same way as the actual hardware model. This is due to the fact that the functional specification is usually translated into HDL which may result in some aspects of the functionality being lost. Moreover, the HDL version of the module becomes the focus of attention and any changes made to it are not implemented in the functional specification. Therefore, by using C for SoC verification, the designer runs the risk of incorrectly modeling the interaction between various SoC components.

These problems can be alleviated through the use of a common language to specify both the behavioral and hardware representations of the module. This can be done by using SystemC. SystemC is an extension to C/C++ which includes constructs to model hardware behavior. The use of SystemC for co-modeling provides a more accurate picture of the behavior of the system and the interaction between SoC components. Most SystemC development to date has focussed on system modeling in software rather than on integration of SystemC modules with logic emulators and simulation accelerators. Given the advanced complexity of many SoC components and the need to verify system level interaction at distinct points in the design cycle, the need for a SystemC interface to verification hardware is evident.

2.4 SystemC

2.4.1 Introduction

SystemC is a C++ class library and a methodology that can be used to effectively create a cycle accurate model of software algorithms, hardware architecture, and interfaces between SoC components. SystemC constructs can be used with standard C++ compilers and debuggers to create a system level model. This model can be used to provide hardware and software development teams with an executable specification of the system. This specification is often a C/C++ program that exhibits the same behavior as the proposed system.

C or C++ provide the control and data abstractions necessary to develop compact and efficient system descriptions. A large number of development tools are associated with these popular programming languages. Standard C and C++ lack the necessary constructs to model hardware behavior such as timing, concurrency, and reactive behavior. SystemC provides a solution to this limitation by offering hardware constructs in the form of C++ classes. A class based approach to providing modeling constructs is superior to a proprietary new language because it allows designers to continue to use the language and tools with which they are familiar. Unlike proprietary solutions, SystemC is open source and any changes or additions are standardized by the Open SystemC Initiative. SystemC includes a cycle based simulation kernel that supports clock based hardware modeling at the system level, behavioral level and register transfer level [10]. Cycle based C simulations are much faster than software HDL simulators.

2.4.2 Hardware Modeling with SystemC

The flowchart of Figure 2.3(a) shows the current system design methodology. A C or C++ model of the system is written to verify concepts at the algorithmic level. After validation is complete, parts of the model are manually converted to a hardware

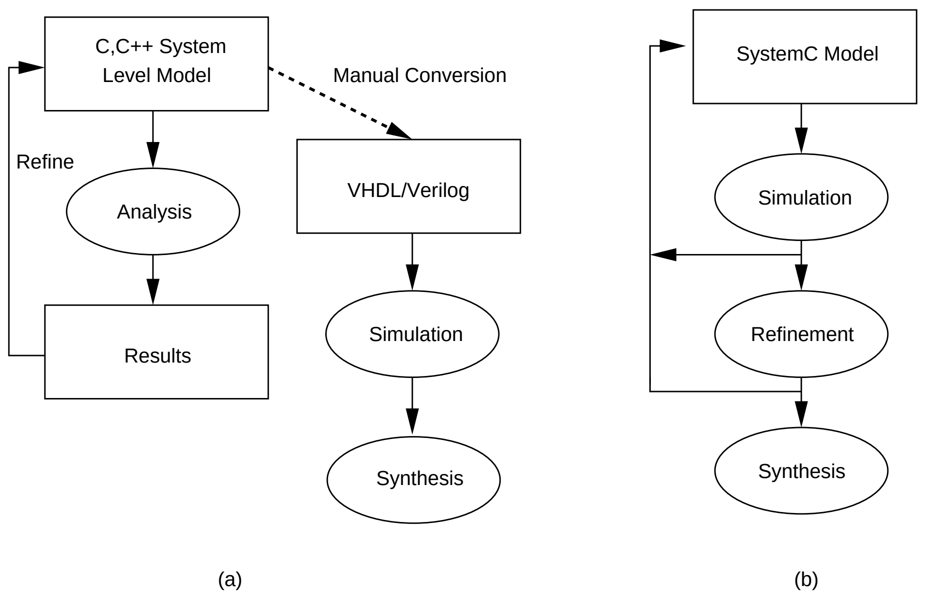


Figure 2.3 System Design Methodologies

description language (HDL), like Verilog or VHDL, for hardware implementation. Unfortunately, this approach can lead to numerous problems. Manual conversion can cause errors. Tests that validate the C model cannot be used to validate the HDL model often leading to a second testbench.

When a SystemC based design methodology is used, the entire system including software and cycle accurate hardware can be specified in one language. The result is the standardization of all design information and the capability to quickly debug, re-specify or re-model design changes. The flowchart of Figure 2.3(b) shows a system design methodology which uses SystemC.

2.4.3 A Simple Example

The following code examples show how to model a D flip flop with an asynchronous reset in Verilog, VHDL and SystemC.

The VHDL model can be written as follows :

```
library ieee;
use ieee.std_logic_1164.all;
entity dffa is
port (
clock : in  std_logic;
reset : in  std_logic;
din   : in  std_logic;
dout  : out std_logic;
     );
architecture rtl of dffa is
begin
   process(reset,clock)
   begin
      if reset = ’1’ then
         dout <= ’0’;
      elsif clock’event and clock = ’1’ then
         dout <= din;
      end if;
   end process;
end rtl;
```

The Verilog model can be written as follows :

```
module dffa(clock,reset,din,dout);
input clock,reset,din;
output dout;
reg dout;
always @(posedge clock or reset)
begin
   if (reset)
      dout <= 1’b0;
   else
      dout = din;
end
endmodule
```

The corresponding SystemC implementation is shown below :

```
#include ‘‘systemc.h’’
SC_MODULE(dffa)
{
   sc_in<bool> clock;    // Input port
   sc_in<bool> reset;    // Input port
   sc_in<bool> din;      // Input port
   sc_out<bool> dout;    // Output port

   void do_ffa()         // Process
   {
      if (reset)
      {
         dout = false;
      }
      else if (clock.event())
      {
         dout = din;
      }
   };

   SC_CTOR(dffa)  // Module constructor
   {
      SC_METHOD(do_ffa);
      sensitive(reset);
      sensitive_pos(clock);
   }
};
```

A **module** is the basic class for objects in SystemC. Modules allow designers to break large complex designs into smaller more manageable pieces. Modules are declared with the SystemC keyword, *SC_MODULE*. A large design will typically be divided into a number of modules that represent logical areas of functionality of the design. Modules communicate with other modules via **ports**. In the above example, the D flip flop has three input ports (*clock,reset and din*) and one output port(*dout*). Ports are specified with the keywords, *sc_in* for input ports, and *sc_out* for output ports. **Signals** connect ports together. They represent the physical wires that interconnect devices. Signals carry data while ports determine the direction of data. **Processes** provide functionality to the module. Processes communicate with each other via signals, and explicit *clocks* can be used to order events and synchronize processes. The above model has one process, *do_ffa()* which performs the functionality of the D flip flop. The process checks the value of the *reset* port. If it is a '0' then the output port is assigned a value '0'. Otherwise, if a positive edge of the clock occurs, the output just reflects the input. Every module must have a constructor to initialize the module. Constructors are specified with the keyword *SC_CTOR*. Besides initializing values for the module variables, the constructor also sets the sensitivities of the various processes in the module. In the above module, process *do_ffa()* is sensitive to two signals - *reset* and the positive edge of *clock*. This indicates that *do_ffa()* will be executed whenever the value of *reset* changes, or a positive edge occurs on *clock*.

2.5 Related Work

Heterogenous co-simulation environments between C and Verilog/VHDL already exist. Ikos Systems has developed a simulator, called TIPSIM, for their TIP co-modeling environment. TIPSIM simulates the functionality of the emulator and the DUT using Verilog models. A C program is used to model the testbench. Communication between software (C models) and hardware (Verilog models) is implemented through the Verilog Programming Language Interface (PLI) using sockets. There is an overhead in passing data back and forth between the HDL based hardware domain and the C/C++ based software domain. This overhead can be reduced but cannot be eliminated. The main purpose of TIPSIM is to test the functionality of the user DUT and transactors in software before implementing them on the emulator.

A different approach to co-simulation is presented in [3] in which SystemC is used to describe both hardware and software components of a system. This approach offers better performance than TIPSIM since it does not use any type of PLI mechanism for data transfer. Moreover, hardware can be synthesized directly from SystemC, which eliminates the need for translation to an HDL. This not only reduces translation time but eliminates bugs introduced during translation. This approach enables designers to perform hardware-software co-verification at very early stages of the design. Various architectures may be explored since the co-simulation process is very fast. As the system becomes more and more refined, hardware can be implemented in gates using synthesis tools and compilers can be used for software. This approach is shown to be three times faster than traditional HDL based co-simulation due to the simplification of the communication between the hardware and software domains. However, it is useful only in the earlier stages of the design cycle when the system specification is still flexible and various design alternatives are being explored.

It is clear that another method of co-verification will be required when the hardware has advanced to the gate level. Logic simulation could be used, but simulation performance will be very slow. Logic emulation along with co-modeling offers a high speed solution. However, a new stimulus environment has to be created in order to use logic emulation and this limits its adoption despite the significant performance gains that can be achieved. A transaction based architecture which provides for 100% portability of a C based testbench for simulation and emulation is presented in [5]. The main focus of this work is to re-use the stimulus driving environment for both simulation and emulation.

The problem of using emulation to its full potential when used in a SoC verification environment remains largely unsolved. Recently, Ikos Systems has presented a proposal called the Standard Co-Emulation Modeling Interface (SCE-MI)[11]. This proposal aims to create a standardized C/C++ modeling interface for emulators and other verification platforms. Its aim is to provide multiple channels of communication that allow software models describing system behavior to connect to structural models describing implementation of a DUT. SystemC has been chosen as the C/C++ modeling environment since it is ideally suited for both untimed and cycle accurate modeling. An infrastructure has been created that allows an untimed functional model to exchange data with a cycle accurate DUT implemented on a simulator or an emulator.

CHAPTER 3

RESEARCH METHODOLOGY

3.1 Research Goals

The main objective of this research is to develop an interface between SystemC and the emulator and to demonstrate its capability to effectively model the interaction between various components of an SoC. This is demonstrated by creating a typical SoC design involving a Reed Solomon coder, a Viterbi decoder, and an interleaver/de-interleaver. The Reed Solomon coder is implemented on the emulator. The Viterbi decoder is implemented as a software C model. The interleaver/de-interleaver which is required to re-order the data exchanged between the Reed Solomon coder and the Viterbi coder is modeled in SystemC. The net result is a system that contains components modeled in 3 different environments - SystemC, C and RTL. In the process of achieving this goal, the functionality of a Reed Solomon coder is verified by means of a co-modeling application. This demonstrates speed gains that can be achieved by using the emulator instead of a software-based HDL simulator.

Another goal of this research is to explore testbench migration. Since the host workstation - emulator interface is a bottleneck to verification speed, we migrate testbench vectors to the emulator to accelerate verification.

3.2 System Architecture

Figure 3.1 shows the elements of our system and how they are interconnected.

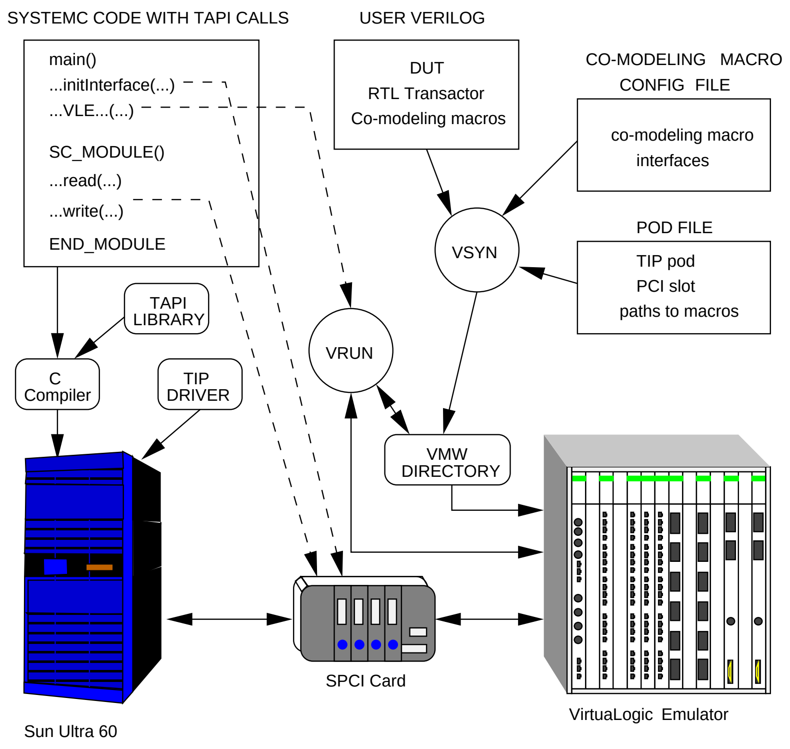


Figure 3.1 System Setup

The user's code runs on a Sun Ultra 60 workstation. It contains SystemC constructs in addition to TAPI function calls. The application is linked with the TAPI and SystemC libraries and compiled. Physical communication with the emulator takes place through an SPCI card. The user application is able to communicate with the card using the TIP driver installed on the host workstation. On the emulator side, the user's RTL code contains an instantiation of the DUT, the RTL transactor and the co-modeling macros. These are compiled and synthesized onto the emulator by emulator system software called VSYN. The VRUN component of the emulator system software controls loading of the design onto the emulator and its operation. Both VRUN and VSYN components are located on the host workstation. The VMW directory is a location on the host workstation where the design files are stored. The co-modeling macro configuration file specifies the interfaces of the co-modeling macros. It contains the bit widths and terminal names of the input and output signals for the various co-modeling macros. The pod file maps a particular set of co-modeling macros to a particular pod on the emulator. A pod is the interface through which the emulator is connected to the host workstation using a PCI cable. It establishes a logical connection through which the macros and the TIP interface exchange data.

3.3 DUT Verification

3.3.1 DUT Overview

The design implemented on the emulator is a fully programmable Reed Solomon (RS) [1] encoder/decoder. This core contains approximately 40k gates and was obtained from Texas Instruments. The design is well suited for emulation since several requirements are met:

- Fully synchronous design style
- Uses a single clock
- Verification based on a large set of stimuli

3.3.2 Simulation

All clock and control information is provided in the test vectors. An example test vector is shown in Figure 3.2. It consists of two portions - an input vector portion and an output vector portion. The input vector consists of 34 bits and includes values for clock (1 bit), reset (1 bit), address (12 bits), control signals (4 bits) and input data (16 bits). The address bits and input data bits are represented in hexadecimal notation. The testbench converts these values into their binary equivalent and applies them to the appropriate DUT inputs. Each output vector consists of 22 bits and includes values for output data (16 bits) and status signals (6 bits). Output data values are represented in hexadecimal notation. The testbench is responsible for converting the output data bits from the DUT into hexadecimal values and writing them into a file.

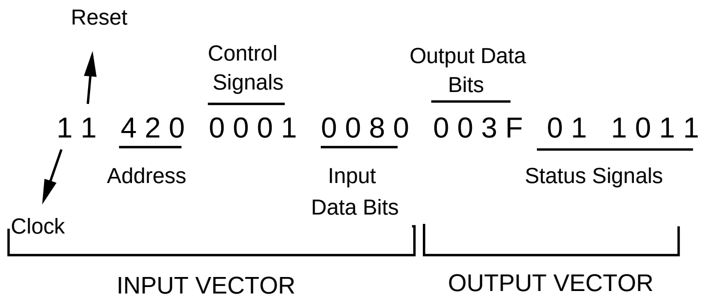


Figure 3.2 A Sample Test Vector

The simulation scenario with test vectors is depicted in Figure 3.3. Input test vectors were applied to the DUT and output result vectors were collected from the DUT by a VHDL testbench. The testbench is responsible for applying input test vectors and collecting output result vectors. The DUT is a gate level netlist of the Reed Solomon coder that is obtained by synthesizing an RTL description provided by Texas Instruments. Synopsys Design Compiler [34] was used for RTL synthesis. During simulation, vectors are read from files using the *TEXTIO* package. The *TEXTIO* package is provided by the VHDL IEEE standard logic library and provides file input/output capabilities to VHDL models. The output vectors from the DUT are written to a file and are compared with the expected results using a comparison script written in Perl, shown in Appendix C. Test vectors and expected outputs were provided by Texas Instruments. Both the testbench and the DUT are simulated on the workstation as a single process. Testbenches of varying sizes, shown in Table 3.1, are used to verify the functionality of the core.

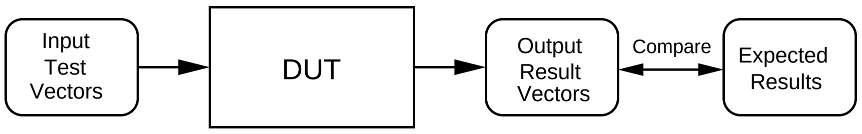


Figure 3.3 Simulation Environment

Table 3.1 shows the time taken to simulate each RS coder testbench. These times are reported by the simulator after the completion of each verification run. All runs were performed on an unloaded Sun Ultra 60 workstation with a single UltraSparc-II 360MHz CPU and 512MB of memory. The Affirma NC VHDL simulator from Cadence was used for simulation. The times shown in Table 3.1 include the time taken to send the test vectors to the DUT, the time taken by the DUT to process the test vectors, and the time taken to receive the output result vectors from the DUT. It does not include the time needed to compare the actual output with the expected output. This time can be considered as a constant for a given testbench size.

Testbench	Number of vectors	Time(sec)
T1	61714	50
T2	68066	55
T3	128270	113
T4	170594	126
T5	179804	134
T6	275262	211

Table 3.1 Time taken for simulation of various testbenches

3.3.3 Emulation

An Ikos VirtuaLogic VLE-2M IDS emulator was used as a second approach for verification. Figure 3.4 shows the design flow used to perform verification using the emulator. The first step in the flow defines the interface between the C model on the host workstation and the DUT on the emulator. This involves designing the protocol used by the emulator and the host workstation to exchange data. Since data streaming is used, there is no need to use a pre-defined protocol. It is enough to specify the format of the data being exchanged, and the bit widths of the communication interface. The data format is similar to the test vector shown in Figure 3.2. Clock information is not provided in the test vector for emulation. The clock generated by the emulator is used for this purpose. The input vector is 33 bits long and the output vector is 22 bits long. More implementation details are shown in Table 3.2. The interface must be designed so that it minimizes the number of transactions between the emulator and the host workstation. This ensures maximum performance. This can be done by selecting proper input and output vector widths for the interface. The most efficient transactions take place across a 512 bit (for a double edge positive and negative clocked design) or 1024 bit (for a single edge clocked design) interface [25].

Since data streaming was used, the number of transactions across the interface equals the number of test vectors in the testbench.

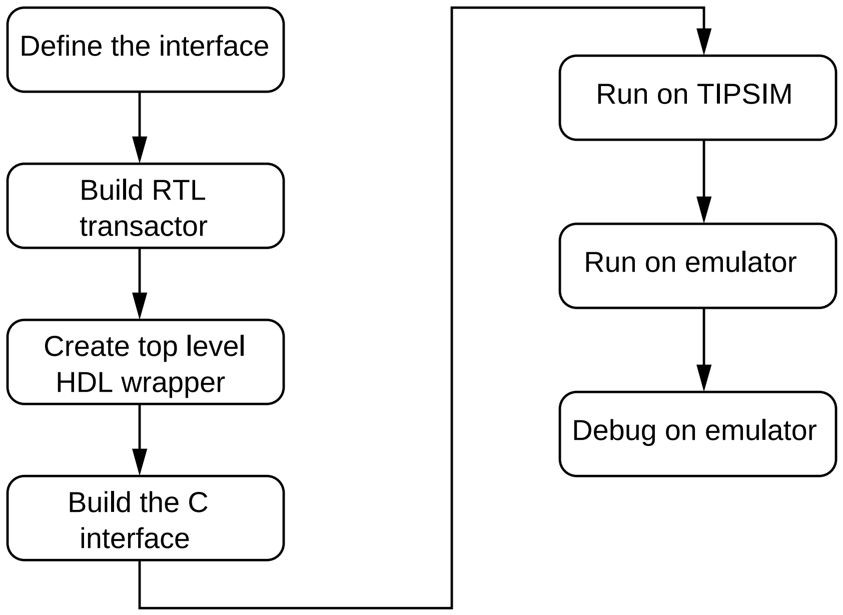


Figure 3.4 TIP/Emulator design flow

Once the interface was established, the RTL transactor was written and tested. The RTL transactor manages flow control into and out of the emulator environment. For emulation of the RS coder, the standard data streaming transactor supplied by Ikos Systems was used. Figure 3.5 shows the state diagram for the transactor. Figure 3.6 shows how the transactor interfaces with the co-modeling macros and the DUT.

The transactor is a two edge vector transactor. The transmit/receive operations are performed on the positive edge of the clock and receive/transmit acknowledgement is performed on the negative edge of the clock. There are four possible transactor states: Idle, Active, Rcvwait and Txwait. When the reset co-modeling macro is called, the Idle state is asserted. From Idle, the next state is Active. Ultimately, the system executes a complete receive/transmit cycle and returns to the Idle state. This can be accomplished in a single cycle if both receive and transmit operations are successful. Otherwise a wait state (Rcvwait or Txwait) is entered. If a wait state is reached, no transfers can be completed until the Idle state is restored through the completion of a receive or transmit cycle. For example, if only a receive is successful, the Txwait state is entered and only when the transmit is successful (by assertion/de-assertion of the handshaking signals by the co-modeling macros), the Idle state is restored. The signals *in_avail* and *out_done* refer to the output handshaking signals from the input and output co-modeling macros respectively. Figure 3.6 shows how these signals are asserted and de-asserted by the transactor and co-modeling macros. The handshaking process is explained in Section 2.3.2.

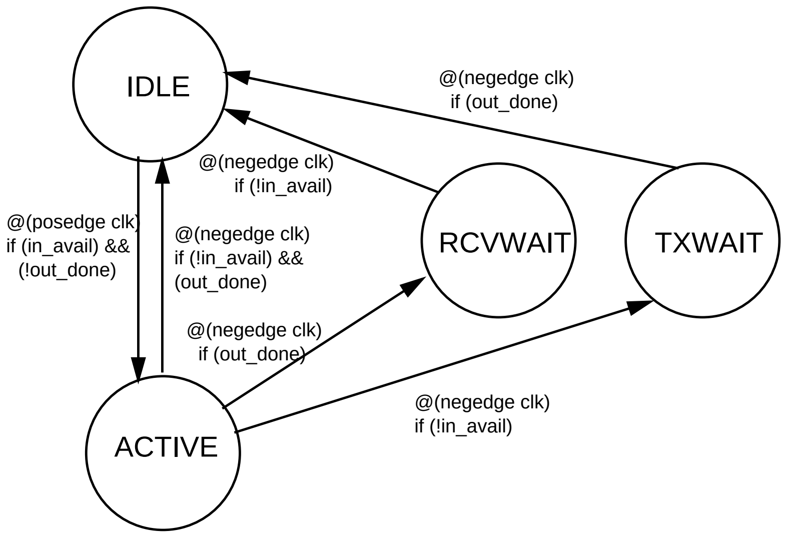


Figure 3.5 State Diagram for Vector RTL transactor

After the RTL transactor was designed, a top level HDL netlist was created. This netlist contains instantiations of the DUT, the RTL transactor and co-modeling macros. The inputs to the DUT are wired to the outputs of the input co-modeling macro and the outputs of the DUT are wired to the inputs of the output co-modeling macro. Since data streaming was used, the DUT I/Os do not need to be routed through the RTL transactor. The transactor must interface with the input and output co-modeling macros to control the flow of data into and out of the DUT. Figure 3.6 shows how components are connected in the top level DUT netlist. The dashed lines indicate the flow of the input and output vectors in the data streaming case.

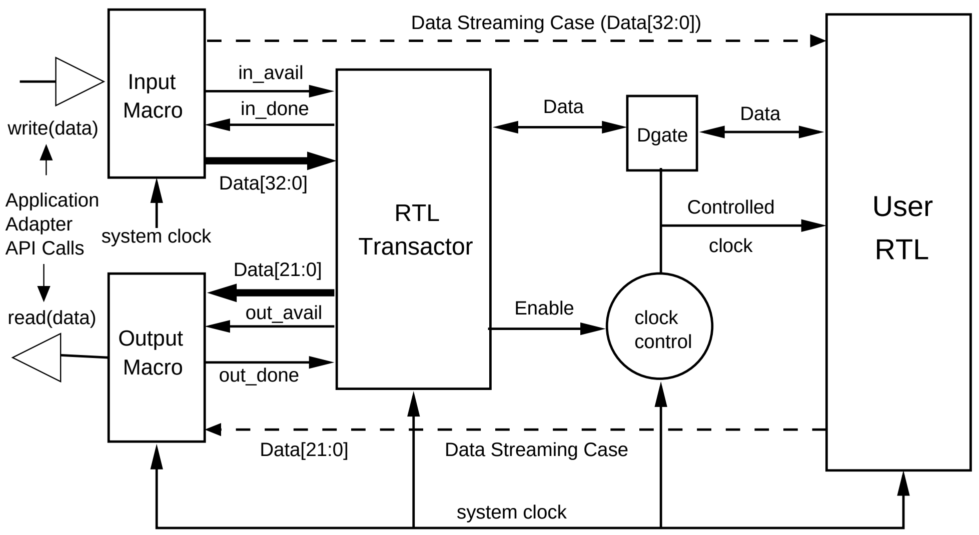


Figure 3.6 Top level HDL netlist block diagram [5]

The next step in the verification process was to create a C testbench to interact with the co-modeling macros using TAPI function calls. These function calls are provided by the C adaptor and are described in Section 2.3.3. It is not necessary for the C adaptor and the C testbench to be two distinct components. They can be merged into the same piece of code if necessary. Design performance depends on how the C testbench is implemented. As a general rule, file I/O operations should be limited during the co-modeling run. If possible, the input test vectors should be buffered into a memory array from disk before the co-modeling run starts and output result vectors should be buffered into a memory array before the data is written to disk.

TIPSIM [25], a tool which simulates the co-modeling interface and emulator using a software simulator, was used to debug the transactor. Figure 3.7 shows how co-modeling is implemented using TIPSIM modules in software. It is almost identical to the TIP hardware system architecture diagram shown in Figure 2.1. In this case, the emulator and its interfaces are simulated in software on a workstation. Transactions between the user application and the simulated emulator interface are performed by means of PLI calls to a separate simulator process from C, instead of direct calls to drivers for a PCI card. TIPSIM facilitates rapid debug of the transactor since there is no need to perform time consuming FPGA compiles for the emulator.

Following verification with TIPSIM, the design was compiled onto the emulator using VSYN, the emulator system software. The compiler takes the input Verilog netlists and partitions them amongst the FPGAs. The compiler then performs time domain multiplexing of FPGA interconnect to overcome pin limitations [4]. The result is a pipelined and multiplexed implementation of inter-FPGA signal paths. As a final step, FPGA place and route is performed and configuration bitstreams are created. The VSYN compiler requires the input netlist to be in the form of gate level primitives belonging a special library called the Virtual Machine Works (VMW) library. The VMW library was provided along with the emulation system software.

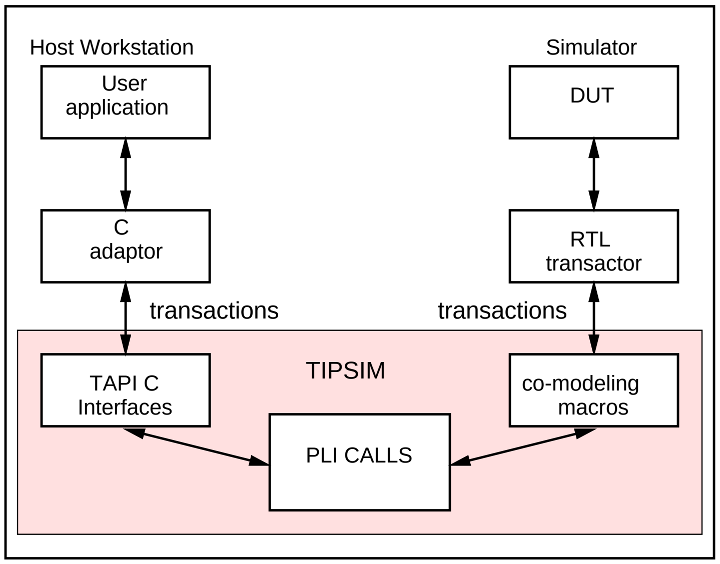


Figure 3.7 Block diagram of TIPSIM interface

Design Compiler from Synopsys [34] was used to synthesize the design into a gate level netlist using the library primitives. Table 3.2 shows the various implementation details for the DUT.

Number of FPGAs (XC4036)	10
Emulator Speed	30 MHz
Design Speed	545 kHz
Input Vector Width	33 bits
Output Vector Width	22 bits

Table 3.2 DUT Implementation Details

3.3.4 Memory Modeling

To allow for memory emulation, functionality of the memory module used in the design netlist must be mapped to the physical memory available in the emulator. The VirtuaLogic emulator used for this thesis contains 32 single-ported, 64K x 32 asynchronous memories per board with read/write enables controlled by adjacent FPGAs. If the functionality of the design memories and the emulator memories do not match, a *wrapper* netlist, which logically maps a design memory onto the emulator memory, is required. The wrapper converts memory control signal sequencing so that a more complicated memory can be emulated on a simpler one. Since the memories used in the Reed Solomon coder were clocked, a one to one mapping between the design memory and the emulator memory was not possible. This required the creation of a wrapper netlist for the Reed Solomon memories.

In Figure 3.8, Mem1 is a memory that is used by the RS coder. The memory has an address port (A), a write data port (D), a read data port (Q), two enable signals (EZ and WZ) and a clock signal (CLK). Mem2 is a memory module that can be converted by emulation software into a physical implementation on the emulator. Mem2 is a single ported memory with read and write enable signals (REN and WEN), an address port (ADR), and a data port (DATA).

An example wrapper implementation to perform signal conversion is shown below. The complete wrapper code is shown in Appendix D.

```
// Assert WEN when EZ, WZ are low and CLK is high
always @(posedge CLK)
begin
   if ((!EZ)&&(!WZ))
      WEN <= 1’b1;
   else
      WEN <= 1’b0;
```

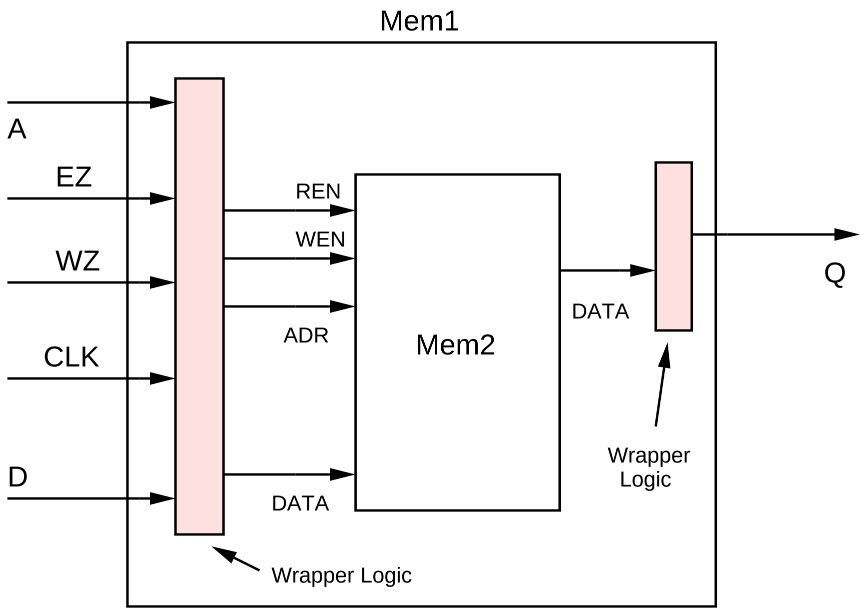


Figure 3.8 Memory wrapper

```
end

// Connect D to DATA when EZ, WZ are low and CLK is high
always @(posedge CLK)
begin
   if ((!EZ)&&(!WZ))
      DATA <= D;
end

//  Connect A to ADR when EZ, WZ are low and CLK is high
always @(posedge CLK)
begin
   if ((!EZ)&&(!WZ))
      ADR <= A;
end

// Connect DATA to Q when EZ, WZ are low and CLK is high
always @(posedge CLK)
begin
   if ((!EZ)&&(!WZ))
      Q <= DATA;
```

```
end
```

Wrapper logic was added to the input and output ports of Mem2 to co-ordinate data transfer based on signal values and time sequence. For example, Mem1 goes into write mode when CLK is high, EZ is low and WZ is low. In such a case, the write enable of Mem2 (WEN) must be asserted and the values on ports A and D of Mem1 must be passed to ADR and DATA respectively. Eleven RS coder memories were mapped onto the emulator in this manner.

3.3.5 Co-Modeling

Co-modeling runs were performed for the Reed Solomon coder testbenches shown in Table 3.3. Results of these runs are shown in the table. Emulation times in column 3 and 4 include the time taken to send all input test vectors to the DUT, the time taken by the DUT to process the input vectors, and the time taken to receive the output result vectors from the DUT. These results do not include the setup time for the emulator or time required to download the DUT onto the FPGAs. The time needed to compare the test vectors with the expected output was measured separately and was determined to be between 3 to 4 seconds. All times were measured using the *time()* function calls available in the standard C library. It can be seen that emulation is about 100 times faster than software simulation on average. All tests were conducted on a Sun Ultra 60 workstation with a single UltraSparc-II 360 MHz CPU and 512MB of memory. Although the emulator runs several orders of magnitude faster than the simulator, data I/O through the TAPI interface can be a limiting factor. This issue is explored in more detail in Chapter 4.

Testbench	Number of vectors	Emulation (sec)	Emulation + buffering(sec)	Simulation (sec)
T1	61714	9.627	0.499	50
T2	68066	10.756	0.538	55
T3	128270	20.375	1.022	113
T4	170594	27.014	1.34	126
T5	179804	27.74	1.41	134
T6	275262	42.93	2.15	211

Table 3.3 Times taken for emulation without and with buffering

Most of the co-modeling run time on the emulator was spent reading vectors from the input vector file on the workstation and writing received vectors from the DUT into an output file on the workstation. When the code was profiled (using gprof) and executed, it was revealed that 90% of the execution time was spent in the routines that perform file I/O. The user application was subsequently modified to buffer the entire set of test vectors into an array before sending them to the DUT. Similarly, the output vectors from the DUT were buffered into an array and written into a file at the end of the co-modeling run. As shown in Table 3.3, vector buffering improves emulation performance by about a factor of 20 versus an unbuffered approach.

3.4 SoC Implementation

3.4.1 Overview

In order to test the SystemC-emulator co-modeling interface, a communication system model was created. Figure 3.9 shows a block diagram of the system. Reed Solomon codes [1] can be combined with convolutional codes (such as Viterbi [2]) to form a communication system. In a convolutional code, the outputs for a particular time unit not only depend on the set of inputs received during that time unit, but also on the set of inputs received during a previous time span. Convolutional codes can correct both random bit errors and burst errors. A burst error is a continuous group of random bit errors occuring intermittently in a data bitstream. In this communication system, if the error correcting capability of the Viterbi decoder is exceeded, the decoder output can contain a series of burst errors. Reed Solomon codes are used to correct these errors.

Encoding in our system is performed by passing the source data through a Reed Solomon encoder [20] followed by a Viterbi encoder [15]. The Reed Solomon encoder takes a block of data symbols and appends a set of parity symbols. Reed Solomon codes operate on blocks of data bits, while the Viterbi encoder/decoder operates on a stream of data bits. An interleaver [17] is used to convert the output data from the Reed Solomon encoder (which is in blocks) to a stream of data bits which can be sent to the Viterbi encoder. Decoding is performed by passing the data bitstream through a Viterbi decoder [15] followed by a Reed Solomon decoder [20]. The Reed Solomon decoder uses the parity symbols added by the Reed Solomon encoder to correct errors in the received data block. A de-interleaver [17] is used to convert the data from a stream of bits into a block format which is suitable for the Reed Solomon decoder. To simulate noise, additive white Gaussian noise (AWGN) is assumed for the communication channel.

The concatenated communication system offers a low bit error rate (BER) which is desirable for many communication applications. One specification of this system is the Consultive Committee for Space Data Systems (CCSDS) [13] recommendation

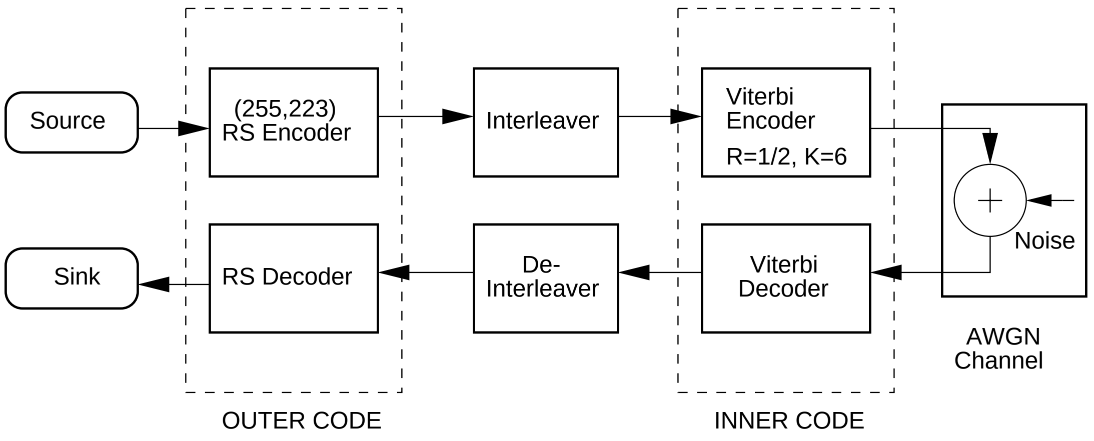


Figure 3.9 Block diagram of the CCSDS system

which uses a (255,223) Reed Solomon code and a convolutional code of rate $R = 1/2$ and a constraint length of $K = 6$. This specification has been adopted for use by numerous planetary spacecraft such as NASA's *Cassini* spacecraft to Saturn [17]. An (n, k) RS code indicates that the encoder takes k data symbols and adds $(n-k)$ parity symbols to get n codeword symbols. For example, a (255,223) RS code would take 223 data symbols and add 32 parity symbols to get a codeword of 255 symbols. The rate of the code is the number of output bits that are generated for each input bit. A rate of 1/2 implies that for each input bit, 2 output bits are generated. Constraint length refers to the number of times each input bit has an impact on the output bit. In this case, $K = 6$ implies that each input bit impacts the value of each subsequent output bit 6 times.

The implementation of the concatenated coding system uses a new feature of SystemC 1.2Beta called the master-slave communication library [26]. A brief introduction is presented in the following section.

3.4.2 The SystemC Master-Slave Communication Library

The master-slave communication library can be used to describe systems that use bus communication protocols. Systems that contain DSPs, custom ASIC cores and processor cores communicating over a set of buses are particularly well-suited to this library. The library introduces semantics for sequential execution and communication between processes (the semantics are described later in this section), which supports the functional modeling of software-software, software-hardware and hardware-hardware interfaces. Using this library, systems can be modeled as an interconnection of sequentially communicating functional blocks. As a result, a system can be described in software at the following levels of abstraction.

- Untimed functional level (UTF)

  This level is used to create an executable specification of the system. The system is decomposed into functional modules that communicate over abstract channels in a sequential nature. At this level, data transactions and execution order are modeled accurately, but time is not. All processes execute in zero time but in a well-defined order. The UTF level is primarily used for design exploration and allows for rapid execution.

- Timed functional level (TF)

  When modeling at the timed functional level, component functionality is assigned a fixed runtime which is measured in absolute time units. Time specification is performed in SystemC by the use of *wait(delay)* statements, where *delay* represents a system constraint on the execution time of a module or a time budget. This level is primarily used for performance modeling, hardware-software partitioning and resource allocation since it can be integrated into higher levels of abstraction. The modeled system is not clocked. Inter component communication remains unchanged from the untimed functional level.

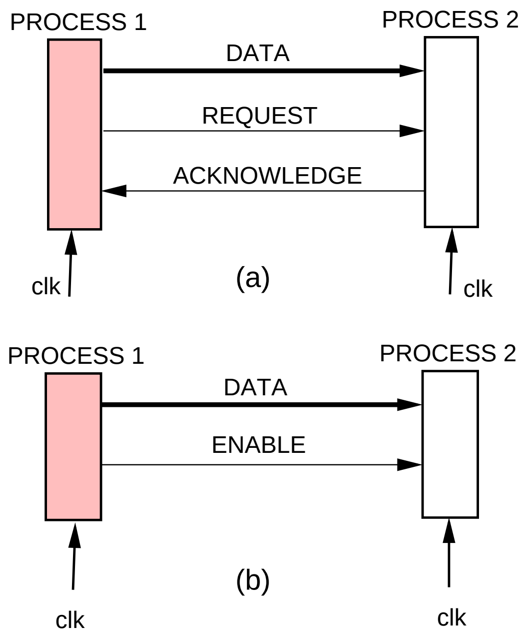


Figure 3.10 (a) Full handshake protocol and (b) Enable handshake protocol

- Bus-cycle accurate level (BCA)

The bus cycle accurate level involves refining synchronous inter-component interaction with respect to bus protocols. Component functionality is modeled at the timed functional level. Abstract ports at the UTF and TF levels are implemented as protocol-based bus ports. A bus port is a hierarchical entity that groups together specific terminals for data, address and control signaling for a bus protocol. SystemC implements three bus protocols - no handshake, enable handshake and full handshake. Figure 3.10(a) shows the terminals of the full handshake protocol.

The data transfer cycle for full handshake proceeds as follows for two software processes. Process 1 asserts *REQUEST* and sends the data to Process 2 via the *DATA* signals. When Process 2 receives the data, it asserts the *ACKNOWLEDGE* signal. This cycle is repeated for the next data item that has to be transferred. The handshaking process is synchronous with respect to a system clock that is given to both processes. Figure 3.10(b) shows the terminals of the enable handshake protocol. Each data transfer cycle is preceeded by the assertion of the *ENABLE* signal. There is no acknowledgement sent back by Process 2.

- Cycle accurate level (CA)

  The lowest level of abstraction is the cycle accurate level (CA) in which component functionality as well as inter-component interaction is synchronously co-ordinated with respect to a system clock. This level can be synthesized to the gate level using SystemC synthesis tools.

The master-slave communication model enables systems to be described at various levels of abstraction since it separates module behavior from inter-module communication. This abstraction enables independent refinement of module functionality and communication. Verification modules can be swapped out and replaced with modules described at different levels of abstraction. Communication between modules can be refined without affecting the processing modules themselves. For example, it is possible to specify an abstract communication model at the UTF level and refine it to a FIFO communication link or a bus communication link at the BCA level.

The SystemC master-slave communication library is used to describe inter-component communication at the different levels of abstraction described above. At the UTF and TF levels, functional communication can be described using abstract ports. An example of abstract port implementation is shown in Figure 3.11. At the BCA and CA levels, these ports are augmented to be protocol-based bus ports. An example is shown in Figure 3.12. Each port is described in the next paragraph via an example.

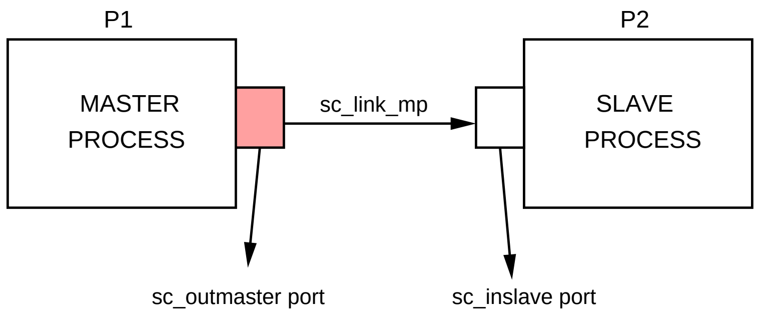


Figure 3.11 Master-Slave communication model for UTF and TF levels

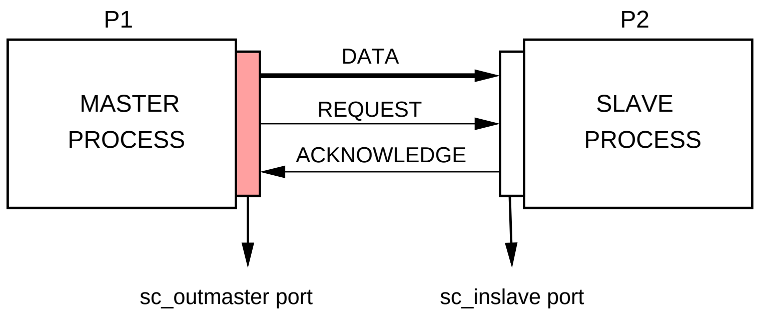


Figure 3.12 Master-Slave communication model for BCA and CA levels with a full handshake protocol

In Figures 3.11 and 3.12, P1 and P2 are two processes which can be in different modules. One of the processes is a *master* process (P1) and the other process is a *slave* process (P2). The slave process has a SystemC port defined by the keyword *sc_inslave*. The master process has a SystemC port defined by the keyword *sc_outmaster*. These two ports are linked by a channel which is defined by the SystemC keyword *sc_link_mp*. These three keywords are provided by the master-slave communication library to faciliate communication between processes. The master process can invoke the slave process by writing a value to its outmaster port. The slave process begins execution when it receives the value at its inslave port. The slave process executes inline with the master process and returns control to the master process after execution. For the BCA and CA implementations, these ports are augmented with handshaking signals shown in Figure 3.12.

3.4.3 Concatenated Coding System Implementation

The communication system described in Section 3.4.1 was implemented as a co-modeling application using both simulation technology and the VirtuaLogic emulator. The Reed Solomon encoder/decoder was implemented on the emulator. Interleaving and de-interleaving was performed using SystemC models. The Viterbi portion of the system, obtained from [15], was implemented as a C model. The interleaver, de-interleaver and Viterbi portions of the system were run on the host workstation as thread processes which communicate with each other via shared memory. A block diagram of the system is shown in Figure 3.13.

A (128,122) RS code was used since the Reed Solomon core does not support the (255,223) code specified by the CCSDS standard. Each component of the sys-

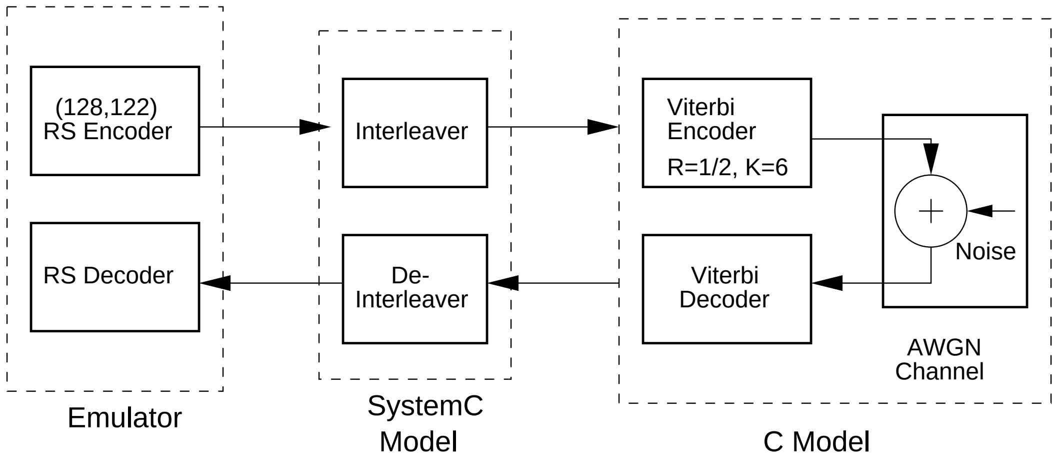


Figure 3.13 Concatenated coding system

tem is represented by a module. In Figure 3.14, RS Encoder, Interleaver, Viterbi Encoder, RS Decoder, De-Interleaver and Viterbi Decoder are the modules of the system. SystemC code for the interleaver and de-interleaver is shown in Appendix B. Modules consist of a number of processes that implement module functionality. For example, the RS Encoder module has two SystemC processes, *emu_setup()* and *rs_encode()* which are responsible for setting up the emulator and performing Reed Solomon encoding respectively. A summary of the modules and their processes is presented in Table 3.4. Communication between the modules is performed via the SystemC master-slave communication library described in Section 3.4.2. Figure 3.14 shows how the master-slave communication model can be applied to the concatenated coding system. The grey boxes indicate master ports and the clear boxes indicate slave ports. The Viterbi module requires a SystemC wrapper which adds the ports required for master slave communication. The SystemC wrapper is shown below.

```
#include "systemc.h"
```

```
SC_MODULE(viterbi)
{
  // Ports added for master-slave communication with interleaver
  // and de-interleaver

  sc_inslave<int> from_int_ctrl;     // Slave port from interleaver
  sc_inslave<sc_bv<2> > from_int_data; // Slave port from interleaver
  sc_outmaster<sc_bv<2> > to_deint_data; // Master port to de-interleaver
  sc_outmaster<int> to_deint_ctrl;  // Master port to de-interleaver

  // Variables local to the Viterbi module

  int quantizer_table[256];
  int *channel_output_matrix;

  // Functions implementing Viterbi encoding and decoding

  void init_quantizer(void)
  void init_adaptive_quant(float es_ovr_n0)
  int soft_met(int data, int guess)
  void code(long input_len,int *in_array,int *out_array)
  float gngauss(float mean, float sigma)
  void initialise()
  void vitdec(int ip1,int ip2,int *out, int flag1)
  void start_all()

  SC_CTOR(viterbi)
    {
      // store_data and start_all are slave processes activated by
      // the interleaver
      SC_SLAVE(store_data,from_int_data);
      SC_SLAVE(start_all,from_int_ctrl);
    }
};
```

The system operates as follows:

1. The emulator is set up and the design is downloaded to the emulator by calling the *emu_setup()* process on the RS Encoder module. This process is called only once at the start of the session.

Module	Processes
RS Encoder	*emu_setup(), rs_encode()*
RS Decoder	*emu_close(), rs_decode()*
Interleaver	*load_data(), read_data()*
De-Interleaver	*store_data(), send_data()*
Viterbi Encoder	*store_data(), encode()*
Viterbi Decoder	*decode()*

Table 3.4 CCSDS system module summary

2. Test vectors stored on the host workstation are applied to the RS encoder (on the emulator) by the *rs_encode()* process. This process also collects the output vectors from the RS encoder and extracts the encoded codewords.

    2.1 Once the encoded codewords have been obtained, they are sent to the interleaver module via the data bus. The process *load_data()* collects the data when it is sent by *rs_encode(). Load_data()* is a slave process of *rs_encode().*

    2.2 When all the codewords have been sent, a signal is sent by *rs_encode()* to the interleaver on the control bus. When the interleaver receives this signal, the *read_data()* process is called. This process performs the interleaving operation and sends the resulting bitstream to the Viterbi module through the data bus. When all the data has been sent *read_data()* sends a signal to the control bus to start Viterbi encoding.

3. The Viterbi encoder module has a slave process, *store_data()* which collects the interleaved bitstream sent by *read_data().* When the module receives the control signal from the interleaver, the bitstream is encoded by the *encode()* process.

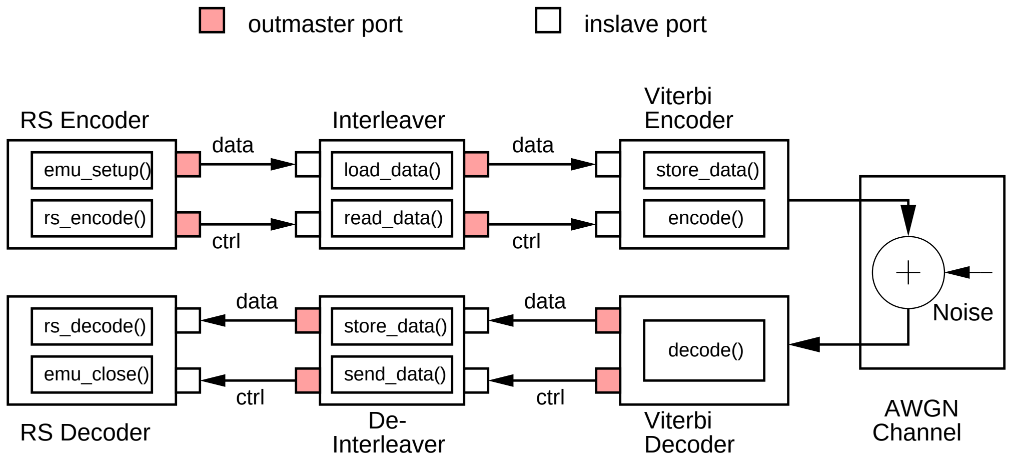


Figure 3.14 Untimed concatenated coding system with master-slave communication

4. Noise is added to the encoded bitstream and the resulting signals are sent to the Viterbi decoder module. The *decode()* process in this module decodes the bitstream and writes the data to be sent to the deinterleaver module to the output data bus. After the data has been sent to the deinterleaver module, a signal is sent on the control bus indicating that the deinterleaver should commence operation.

5. The deinterleaver module has a process, *store_data()*, which receives and stores the data sent by the Viterbi decoder module. When the control signal from the Viterbi decoder module is received, the *send_data()* process is invoked. This process de-interleaves the bitstream and sends the codewords to the RS decoder module through a data bus. After all the codewords have been sent, a signal is sent on the control bus to indicate to the RS decoder module that it can commence operation.

6. The *rs_decode()* process in the RS decoder module sends the codewords to the RS Decoder core on the emulator. When the decoding is complete, another run can commence, or the emulator can be shut down (using the *emu_close()* process)

Once started, the system runs in a sequential manner with point to point communication between the modules. The communication path between modules has been separated into two buses - one each for data and control. Each communication sequence commences with data transmission, followed by the assertion of a control signal which indicates that the module operation can commence. Most of the system is modeled at the untimed functional level with the exception of the RS coder, which is cycle accurate. This model is primarily useful for design exploration.

This system setup offers considerable performance gains over a similar setup implemented completely in software. To explore the performance benefits, a software version of the concatenated coding system was created with the emulated design replaced by a simulated version. The same gate level version of the RS coder was run on both the simulator and emulator. Data was exchanged between the SystemC models and the simulator using TIPSIM (described in Section 3.3.3). The system is similar to the one shown in Figure 3.7. Single value transactions were performed with the emulator. A total of 32500 vectors were sent to the emulator and 32500 vectors were received. The emulator based implementation showed a speedup of 2.3 over the simulator based implementation. The times taken for a single run are shown in Table 3.5.

Further analysis of the results revealed that 16.4 seconds of the total system run time (43 seconds) was spent performing tasks related to the emulator. Out of this

	Emulator(sec)	Simulator(sec)
CCSDS	43	93

Table 3.5 Times taken for emulator and simulator based implementations of CCSDS

time, 14 seconds was spent configuring the emulator and downloading the DUT onto the FPGAs. This indicates that 6% of the total system run time was spent in data transfer between the emulator and the host workstation. The software portion of the system was the limiting factor for verification performance.

Chapter 4

TESTBENCH INTEGRATION

4.1 Performance Benefits

In spite of the performance improvement that TIP offers, the communication interface limits verification performance. Most verification time is spent in input and output test vector transfer with the DUT. If test vectors could be stored on the emulator and applied to the DUT, the need for the host workstation to send test vectors through the communication interface to the DUT could be eliminated. In this model, the software simply acts as a testbench controller which decides when to initiate and terminate the test.

An attractive option is to self contain the testbench on the emulator. Both the input test vectors and expected output vectors are stored in emulation memory. Each test vector is applied to the DUT inputs and the DUT outputs are compared with expected vectors. The comparison is also performed on the emulator. Once all test vectors have been applied and tested, a pass/fail result is sent to the host workstation. This scheme is efficient since only two vectors are sent through the co-modeling interface rather than an input and result for each vector. One vector is sent by the host workstation to the emulator initiating the test process and one vector is sent by the emulator to the host workstation containing a pass/fail result. Implementation details and performance issues are discussed in the following sections.

4.2 Architecture

Figure 4.1 shows the architecture of the system which includes the integrated testbench. The testbench of the DUT is partitioned into two portions. One portion contains the input test vectors and the other portion contains the expected output vectors. Each portion is stored on the emulator in a test vector memory (TVM). The host workstation controls the test process. Once the *initiate_test* signal is asserted, the test vectors in the input test vector memory are applied to the inputs of the DUT at discrete clock edges. When each test vector is applied, the DUT outputs are bitwise compared with the corresponding vector in the output test vector memory. A pass/fail result for the entire set of test vectors is generated and sent back to the host workstation.

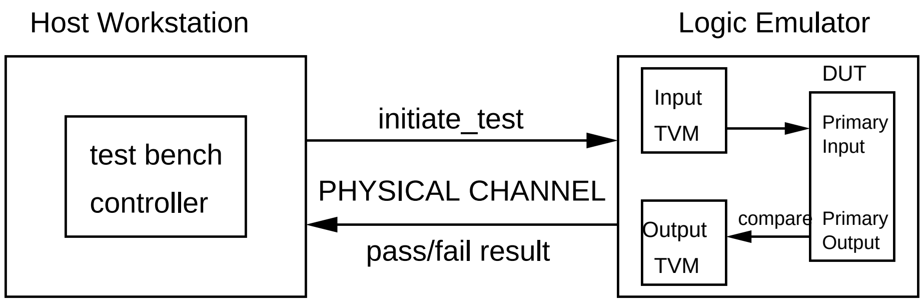


Figure 4.1 Testbench Integration

4.2.1 DUT Modifications

In order for the above approach to work, the DUT architecture had to be modified. A positive edge triggered counter for memory address generation was added to the netlist. The input TVM data corresponding to each address was applied to the inputs of the DUT at every positive clock edge. The outputs of the DUT are compared to the contents of the output TVM corresponding to the same address. The result of each comparison was logically ANDed with previous comparisons to generate a cumulative pass/fail result for the entire set of test vectors. Only when all the vectors match the expected output was a pass result generated. When the counter completed generating all valid memory addresses, the cumulative pass/fail result along with a count of the number of test vectors applied were sent back to the host workstation through a single transaction. Figure 4.2 shows a block diagram of the address generation, checking and interface logic. *IN_OK* and *OUT_VAL* are synchronization signals generated by the counter. Their function is explained in Section 4.2.2. RTL code for the top level DUT is shown in Appendix A.

4.2.2 RTL Transactor

The standard data streaming RTL transactor supplied by Ikos (shown in Figure 3.5) is not suitable for test vector migration since it expects one input vector for every output vector sent by the DUT. Our experiment required a transactor that sends one vector to the DUT, waits for a certain amount of time until the counter has generated all valid memory addresses, and then returns the output vector which contains the pass/fail result. Figure 4.3 shows the changes made to the existing transactor (shown in Figure 2.2) to achieve this behavior and the interaction with the DUT architecture shown in Figure 4.2.

Two synchronization signals *IN_OK* and *OUT_VAL* were added to the top level DUT wrapper. When the DUT wrapper receives the *initiate_test* signal from the host workstation, *IN_OK* and *OUT_VAL* are set to '1' and '0' respectively. When the

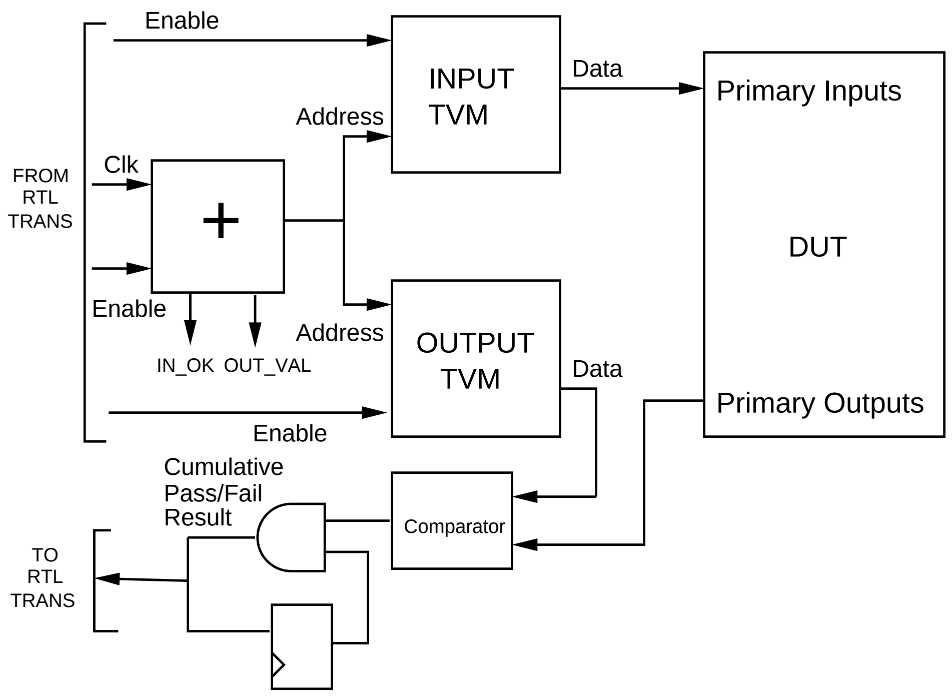


Figure 4.2 DUT architecture modifications

counter has generated all valid memory addresses, the values of the two signals are inverted. Two events must happen for co-modeling to proceed properly. First, the clock to the DUT must be enabled after the first transaction (i.e. the *initiate_test* signal) has been received. This is accomplished by logically ORing the *enable* signal of the RTL transactor with the *IN_OK* signal. The controlled clock will continue to clock the DUT as long as *IN_OK* is high. When the controlled clock is active, the counter generates memory addresses, the contents of the input TVM are applied to the DUT inputs, and the outputs of the DUT are compared with the output TVM contents, at every positive clock edge.

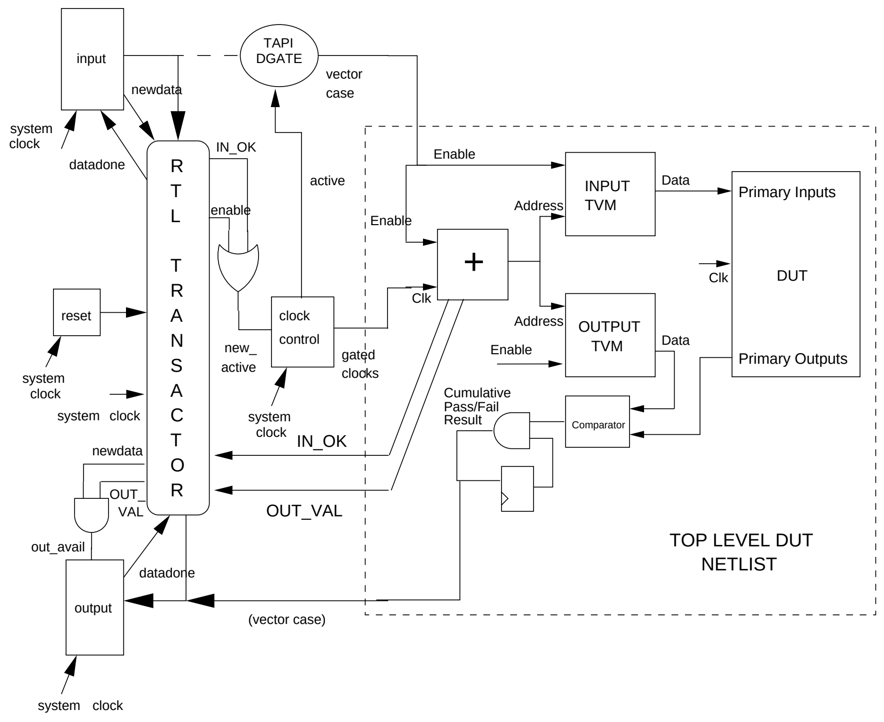


Figure 4.3 RTL transactor modifications

Second, the transactor must inform the output co-modeling macro when the comparision result is ready. This happens when the counter has generated all valid memory addresses. The *newdata* output of the transactor is logically ANDed with the *OUT_VAL* signal from the DUT. The output of the AND gate is connected to the *out_avail* input of the output co-modeling macro. When *out_avail* goes high, the output co-modeling macro will accept data from the DUT wrapper and send it to the host workstation. *OUT_VAL* goes high when the counter has generated all valid memory addresses. This results in the pass/fail result and the transfer of the vector count to the host workstation.

The following C code demonstrates how the testbench on the host workstation inititates the testing process and receives the pass/fail result. In the code, *WriteDVV* is the data structure that contains the value of the vector (in this case the *initiate_test* signal) which needs to be sent to the DUT on the emulator. *ReadDVV* is the data structure that contains the value of the vector sent back by the DUT on the emulator (the pass/fail result and the vector count).

```
// Send the initiate_test signal to the emulator
tapiDVV_setWord(WriteDVV,0,1,&RC);
tapiCOMI1_write(myCOMI, WriteDVV, &RC);

// Check the return code to ensure that the operation
// completed successfully
checkRC(&RC);

// Store the data value sent into a file
print_data_to_writefile(file1, WriteDVV, write_width);

// Wait for a read event from the emulator
// If it is a read event, store the data value sent by
// the emulator in ReadDVV and write it to a file

event = tapiCOMI1_Wait(myCOMI,&RC);
if (RC.type == tapiRC_OK)
  {
    if (event & tapiCOMI1_DataReadable)
      {
        printf("\n Read event received.....");
        tapiCOMI1_read(myCOMI, ReadDVV, &RC);
        checkRC(&RC);
        print_data_to_readfile(file2, ReadDVV, read_width);
      }
  }
else
  {
    printf("Error: RC type not OK.'');
  }
```

4.3 Emulator Implementation

The above approach was implemented with the Reed Solomon coder DUT for varying testbench sizes. Table 4.1 shows the results obtained. Time was measured using *time()* function calls found in the standard C library. The numbers in the last column represent the time taken to process all vectors when the testbench is stored in emulation memory. Columns 3 and 4 represents the time taken when the testbench is stored on the host workstation. Column 3 represents the time to send and receive the test vectors. Column 4 represents the time to compare the received vectors with the expected outputs on the host workstation. The vectors were buffered before being sent to the emulator. It can be seen that the verification performance when the testbench is migrated onto the emulator is 5000 times faster on the average, than when the testbench is kept on the host workstation. This significant performance improvement is obtained since the overhead of sending the test vectors through the communication interface is absent. The numbers in the last column are not linear with the testbench size. Each hardware design which includes vectors requires recompilation before being implemented on the emulator. The compiler assigns different emulation speeds to each of the test cases.

The number of test vectors that can be stored on the emulator depends on the amount of free memory available on the emulator. The largest testbench, T6, did not fit on the emulator due to a lack of memory on the two array boards. Testbench migration also increases the amount of time needed to configure the FPGAs on the array board by a few seconds since the test vectors have to be stored on the emulator.

Testbench	Number of vectors	Workstation(sec)		Emulator(usec)
		Verification	Comparison	
T1	61714	0.499	1.4	795
T2	68066	0.538	1.6	612
T3	128270	1.022	2.65	621
T4	170594	1.34	3.45	706
T5	179804	1.41	3.65	791
T6	275262	2.15	5.55	-

Table 4.1 Times taken for emulation with the testbench on the host workstation and the emulator

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

There is an urgent need for the EDA industry to meet the verification requirements of SoC design teams. While the industry has delivered verification performance in the form of emulation and rapid prototyping systems, the challenge of integration with SoC modeling environments remains unmet.

This thesis presents a methodology for interfacing SystemC with an Ikos emulator. It utilizes the master-slave communication library of SystemC for inter-module communication in software and the Transaction Interface Portal (TIP) for software-emulator communication. A communication system was created with components modeled in C, SystemC and VHDL. A co-modeling application was run to demonstrate the capability of the SystemC-emulator interface to model the interaction between various SoC components. The modeling environment was primarily implemented at the untimed functional level and is a useful tool for design exploration. The emulation-based system was shown to be 2.3 times faster than a similar system implemented completely in software. The performance of the software implementation is limited by the PLI based mechanism used to exchange data between C and HDL domains.

This work also demonstrates the high performance that logic emulators offer over software based simulators. Logic emulation was shown to be about 100 times faster than software simulation when verifying the functionality of the Reed Solomon coder. For many industry ASICs, testbenches include millions of vectors. In such cases, logic emulation is an attractive alternative to software simulation. In this thesis, it is shown that significant improvements in verification time can be achieved by using software techniques such as vector buffering.

In spite of the high performance that can be obtained through co-modeling, data transfer through the co-modeling interface is the limiting factor for verification performance. A technique to improve verification speed by migrating the testbench onto the emulator was presented in Chapter 4. By storing the input test vectors and expected output vectors in memories on the emulator, we eliminated the need to send test vectors through the co-modeling interface. This translates into a substantial speed up in verification performance (5000X), at a cost of increased emulator configuration time due to an increase in downloaded data.

5.2 Future Work

In the communication system described in Section 3.4.3, it would be desirable to refine the SystemC portions of the model to a cycle accurate level using a SystemC synthesis tool such as [35] and implement it on the emulator. The implementation of the testbench migration technique described in Section 4.2.1 is DUT specific and will have to be re-implemented for different DUTs. It would be ideal to obtain a generic implementation which could then be “added on” as a module to any given

DUT. It would also be worthwhile to investigate the feasibility of implementing this functionality within the compiler itself.

APPENDIX A

The following code is the RTL for the top level DUT netlist described in Section 4.2.1). It contains instantiations of the the RS coder, and the testbench migration circuitry (counter, input TVM and output TVM).

```
-- Standard library includes

library ieee;
use ieee.std_logic_1164.all;
use work.std_logic_arith.all;
use work.std_logic_unsigned.all;
use work.copro_header.all;

-- Top level DUT entity. This is the entity that is connected to the RTL
-- transactor (Figure 4.3)

entity top_mem is
  port
    (
      clk       : in std_logic;     -- clock input
      init      : in std_logic;     -- initiate_test signal
      load      : in std_logic;     -- enable signal for memories
      result    : out std_logic;    -- cumulative pass/fail result
      vec       : out std_logic_vector(18 downto 0); -- test vector count
      IN_OK     : out std_logic;    -- synchronization signal
      OUT_VAL   : out std_logic     -- synchronization signal
    );
end top_mem;

architecture structural of top_mem is

-- Reed Solomon coder core

  component rs_coder
  port
    (
    clk           : in  std_logic;
    i1            : in  std_logic;
    i2            : in  std_logic_vector( 10 downto 0 );
    i3            : in  std_logic;
    i4            : in  std_logic;
    i5            : in  std_logic_vector( 4 downto 0 );
    i6            : in  std_logic;         -- 0 write - 1 read
    i7            : in  std_logic_vector( 15 downto 0 );
    o1            : out std_logic_vector( 15 downto 0 );
    o2            : out std_logic_vector(  1 downto 0 );
    o3            : out std_logic;
    o4            : out std_logic;
    o5            : out std_logic;
    o6            : out std_logic
    );
  end component;

-- This is the input TVM (Figure 4.3)

  component test_ram
  port
    (
        readwr      : in std_logic;  -- enable signal
        load        : in std_logic;  -- dummy port
        address     : in std_logic_vector(18 downto 0);  -- address port
        data_out    : out std_logic_vector(35 downto 0)  -- data port
    );
  end component;

-- This is the output TVM (Figure 4.3)

  component check_ram
  port
    (
        readwr      : in std_logic;  -- enable signal
        load        : in std_logic;  -- dummy port
        address     : in std_logic_vector(18 downto 0);  -- address port
        out_vector  : out std_logic_vector(21 downto 0)  -- data port
    );
  end component;

-- Signals to connect to the ports

  -- input TVM data port signal
  signal data_bits        : std_logic_vector(35 downto 0);
  -- counter output
  signal count_out        : std_logic_vector(18 downto 0);
  -- RS Coder output signals
  signal data_out         : std_logic_vector(15 downto 0);
  signal data_out_size    : std_logic_vector(1 downto 0);
  signal enc_complete     : std_logic;
  signal dec_complete     : std_logic;
  signal ready            : std_logic;
  signal data_valid       : std_logic;
  -- output TVM data port signal
```

```
signal out_vector       : std_logic_vector(21 downto 0);
-- concatenated RS Coder output signal
signal compare_vec      : std_logic_vector(21 downto 0);
-- Memory enable
signal readwr           : std_logic;

begin

    -- This process will reset the counter to zero at the start
    -- of the co-modeling session, and will start incrementing
    -- at every positive edge of the clock. The range of memory
    -- addresses generated depends on the testbench size
    -- It has been hardcoded here to 275262

    process
    begin
      wait until (clk'event and clk='1');
        count_out <= "0000000000000000000";
        while (count_out < "1000011001100111110") loop
          count_out <= count_out + '1';
          wait until (clk'event and clk='1');
        end loop;
    end process;

    -- This process will be invoked whenever the output of the
    -- counter changes. It will set values for the synchronization
    -- signals (IN_OK, OUT_VAL), memory enable (readwr), pass/fail
    -- result (result) and the output vector count (vec) depending
    -- on the state of the counter output.

    combine : process(count_out)
      variable temp_result : std_logic;
      variable comp_result : std_logic;
    begin

    -- If the counter output is zero, IN_OK and OUT_VAL are set
    -- to 1 and 0 respectively. Memory is enabled (readwr=1)
    -- and the output vector count is set to zero.

      if (count_out = "0000000000000000000") then
        temp_result := '1';
        IN_OK <= '1';
        OUT_VAL <= '0';
        readwr <= '1';
        result <= temp_result;
        vec <= "0000000000000000000";

    -- If the counter address is still valid (less than the
      -- testbench size), concatenate the outputs of the RS coder,
      -- and compare them with the contents of the output TVM
      -- AND the comparison result with the previous result.

        elsif (count_out < "1000011001100111101") then
          compare_vec <= data_out(15 downto 0) & data_out_size(1 downto 0)
          & ready & data_valid & enc_complete & dec_complete;
          if (compare_vec = out_vector) then
            comp_result := '1';
          else
            comp_result := '0';
          end if;
          temp_result := temp_result and comp_result;
          result <= temp_result;
          vec <= "0000000000000000000";
          IN_OK <= '1';
          OUT_VAL <= '0';
          readwr <= '1';

      -- Counter has generated all valid memory addresses. Complement
      -- the synchronization signals, de-assert the memory enable
      -- and sent the vector count and pass/fail result to the RTL
      -- transactor

        else
          readwr <= '0';
          IN_OK <= '0';
          OUT_VAL <= '1';
          result <= temp_result;
          vec <= count_out;
        end if;
      end process;

      -- Instantiations of Input TVM, Output TVM and the Reed Solomon
      -- Coder. Bit ranges of the input TVM data port (data_bits) are
      -- assigned to the inputs of the RS Coder

      S1:TEST_RAM port map(readwr,load,count_out,data_bits);
      S2:CHECK_RAM port map(readwr,load,count_out,out_vector);
      S3:RS_CODER port map(clk,data_bits(35),data_bits(34 downto 24),
         data_bits(23),data_bits(22),data_bits(21 downto 17),data_bits(16),
         data_bits(15 downto 0),data_out,data_out_size,ready,data_valid,
         enc_complete,dec_complete);

end structural;
```

Appendix B

The following code shows SystemC implementations of the interleaver and de-interleaver modules used in the communication system described in Section 3.4.3.

```
#include "systemc.h"

// Interleaver implementation

SC_MODULE(interleaver)
{
  // Ports of the interleaver

  sc_outmaster<sc_bv<2> > to_viterbi_data;  // Data bus to Viterbi
  sc_outmaster<int> to_viterbi_ctrl;   // Control bus to Viterbi
  sc_inslave<sc_bv<8> > from_rscoder_data;  // Data bus from RS Coder
  sc_inslave<int> from_rscoder_ctrl;  // Control bus from RS Coder

  sc_bv<8> ram[2][128];   // Interleaver memory
  sc_uint<8> temp_integer;
  sc_bv<8> tempval,tempval1;
  sc_bv<2> outreg;
  int i,j,k,l;

// This function accepts data from the RS Coder module and stores
// it in the interleaver memory. It is a slave process of the
// RS Coder module

   void load_data()
     {
       temp_integer=from_rscoder_data;
       ram[i][j]=temp_integer;
       j++;
       if(j>127)
 {
   j=0;
   i++;
 }
     }

// This function performs the interleaving operation. Interleaving
// commences when the RS coder module asserts its control bus.
// Contents of the interleaver memory are read out column wise and
// sent to the Viterbi module on the data bus. Once all the data
// has been sent, the control bus is asserted to indicate that the
// Viterbi module can start executing.

   void read_data()
     {
       cout<<endl<<"Interleaver module activated...."<<endl;

       for(l=0;l<128;l++)
 {
   tempval=ram[1][l];
   tempval1=ram[0][l];
   for(k=7;k>=0;k--)
     {
       outreg=(tempval[k],tempval1[k]);
       to_viterbi_data = outreg;
     }
 }
       // Now that the data has been sent, start the Viterbi process
       cout<<endl<<"Activating Viterbi module...."<<endl;
       to_viterbi_ctrl = 1;
     }

// Module constructor. Defines process types and initializes module
// variables

   SC_CTOR(interleaver)
     {
       // Define slave processes load_data() and read_data()
       SC_SLAVE(load_data,from_rscoder_data);
       SC_SLAVE(read_data,from_rscoder_ctrl);

       i=j=l=k=0;
     }
};


// De-interleaver implementation

#include "systemc.h"

SC_MODULE(deinterleaver)
{
  // Ports of the deinterleaver

   sc_inslave<sc_bv<2> > from_vit_data;  // Data bus from Viterbi
   sc_inslave<int> from_vit_ctrl;  // Control bus from Viterbi
   sc_outmaster<sc_bv<8> > to_rsdecoder_data;  // Data bus to RS Coder
   sc_outmaster<int> to_rsdecoder_ctrl;  // Control bus to RS Coder
   sc_bit ram[2][128*8];   // De-interleaver memory
   int i;

// This function performs the de-interleaving operation.
// Contents of the de-interleaver memory are read out row wise and
// sent to the RS decoder module on the data bus. Once all the data
// has been sent, the control bus is asserted to indicate that the
// RS decoder module can start executing.

   void send_data()
     {
       sc_bv<8> temp_reg;
       int i,j,k;

       i=j=0;
       k=7;

       cout<<endl<<"Sending data to RS Decoder...."<<endl;
       for(i=0;i<2;i++)
 {
   for(j=0;j<128*8;j++)
     {
       temp_reg[k]=ram[i][j];
       k--;
       if (k<0)
 {
   to_rsdecoder_data=temp_reg;
   k=7;
 }
     }
 }

       // Data to decoder has been sent. Run the decoder
       to_rsdecoder_ctrl=1;

     }

// This function accepts data from the Viterbi module and stores
// it in the de-interleaver memory. It is a slave process of the
// Viterbi module

   void store_data()
     {
       sc_bv<2> tempvar;
       tempvar=from_vit_data;
       ram[1][i]=tempvar[0];
       ram[0][i]=tempvar[1];
       i++;
     }

// Module constructor. Defines process types and initializes module
// variables
   SC_CTOR(deinterleaver)
   {
      // Define slave processes send_data() and store_data()
      SC_SLAVE(send_data,from_vit_ctrl);
      SC_SLAVE(store_data,from_vit_data);

      i=0;
   }
};
```

Appendix C

This appendix contains two Perl scripts used in Section 3.3.2 and 3.3.3 to parse the test vectors and generate input and output vector files for use on the emulator. The Perl script shown below processes the testbench file used for RTL simulation into a format suitable for use by the emulator.

```
# Perl script to parse the RTL simulation testbench and generate
# input and output vector files for use by TIP on the emulator

open(INPUT,"encoder");  # input testbench
open(OUTPUT,">input.vectors");  # input test vector file
open(OUTPUT1,">output.vectors"); # output test vector file

# This routine takes each line of the testbench and splits it into
# tokens using space as the delimiter.

while(!eof(INPUT))
  {
    @tokens=split(/ /,<INPUT>);

    @decode="";
    @decode1="";
    @addr="";
    @t1="";

    # This portion separates the clock and reset signals
    # Discards the clock signal and writes the reset signal
    # to an array which is eventually written to the output file

    @t1=split(//,$tokens[0]);
    push(@decode1,pop(@t1));

    @addr=split(//,$tokens[1]);

    # This loop decodes the 3 character hex encoded address field
    # into a 11 bit binary address field

    for ($i=0;$i<3;$i++)
    {
     $dummy=shift(@addr);

                if($dummy eq "0")
```

```
       {
if ($i==0) { push(@decode1,"000");}
else { push(@decode,"0000");}
   }
   if($dummy eq "1")
   {
if ($i==0) { push(@decode1,"001");}
else { push(@decode,"0001");}
   }
   if($dummy eq "2")
   {
if ($i==0) { push(@decode1,"010");}
else { push(@decode,"0010");}
   }
   if($dummy eq "3")
   {
if ($i==0) { push(@decode1,"011");}
else { push(@decode,"0011");}
   }
   if($dummy eq "4")
   {
if ($i==0) { push(@decode1,"100");}
else { push(@decode,"0100");}
   }
   if($dummy eq "5")
   {
if ($i==0) { push(@decode1,"101");}
else { push(@decode,"0101");}
   }
   if($dummy eq "6")
   {
if ($i==0) { push(@decode1,"110");}
else { push(@decode,"0110");}
   }
   if($dummy eq "7")
   {
if ($i==0) { push(@decode1,"111");}
else { push(@decode,"0111");}
   }
   if($dummy eq "8")
   {
   if ($i==0) {push(@decode1,"000");}
       else { push(@decode,"1000");}
   }
   if($dummy eq "9")
   {
   if ($i==0) {push(@decode1,"001");}
       else { push(@decode,"1001");}
```

```
    }
    if($dummy eq "A")
    {
    if ($i==0) {push(@decode1,"010");}
        else { push(@decode,"1010");}
    }
    if($dummy eq "B")
    {
    if ($i==0) {push(@decode1,"011");}
        else { push(@decode,"1011");}
    }
    if($dummy eq "C")
    {
    if ($i==0) {push(@decode1,"100");}
        else { push(@decode,"1100");}
    }
    if($dummy eq "D")
    {
    if ($i==0) {push(@decode1,"101");}
        else { push(@decode,"1101");}
    }
    if($dummy eq "E")
    {
    if ($i==0) {push(@decode1,"110");}
        else { push(@decode,"1110");}
    }
    if($dummy eq "F")
    {
    if ($i==0) {push(@decode1,"111");}
        else { push(@decode,"1111");}
}
    }

    # This portion decodes the control signals and pushes them
    # into the array which is written to the output file (at the
    # end of the script)

    @addr=split(//,$tokens[2]);
    push(@decode,$addr[0]);
    push(@decode,$addr[1]);
    if ($addr[2] eq "1")
    {
        push(@decode,"11111");
    }
    else
    {
        push(@decode,"00001");
    }
```

```
push(@decode,$addr[3]);

# This portion decodes the 4 character hex encoded data value
# into a 16 bit binary encoded data value

@addr=split(//,$tokens[3]);

for ($i=0;$i<4;$i++)
{
 $dummy=shift(@addr);

     if($dummy eq "0")
    {
    push(@decode,"0000");
}
if($dummy eq "1")
{
    push(@decode,"0001");
}
if($dummy eq "2")
{
    push(@decode,"0010");
}
if($dummy eq "3")
{
    push(@decode,"0011");
}
if($dummy eq "4")
{
    push(@decode,"0100");
}
if($dummy eq "5")
{
    push(@decode,"0101");
}
if($dummy eq "6")
{
    push(@decode,"0110");
}
if($dummy eq "7")
{
    push(@decode,"0111");
}
if($dummy eq "8")
{
    push(@decode,"1000");
}
if($dummy eq "9")
```

```
    {
        push(@decode,"1001");
    }
    if($dummy eq "A")
    {
        push(@decode,"1010");
    }
    if($dummy eq "B")
    {
        push(@decode,"1011");
    }
    if($dummy eq "C")
    {
        push(@decode,"1100");
    }
    if($dummy eq "D")
    {
        push(@decode,"1101");
    }
    if($dummy eq "E")
    {
        push(@decode,"1110");
    }
    if($dummy eq "F")
    {
        push(@decode,"1111");
}
    }

    # Write the decoded vector array to the output vector file

    for ($z=0;$z<$#decode+1;$z++)
    {
       print(OUTPUT $decode[$z]);
    }
    print(OUTPUT "\n");

    for ($z=0;$z<$#decode1+1;$z++)
    {
       print(OUTPUT $decode1[$z]);
    }
    print(OUTPUT "\n");

    # This portion of the code process the output vector portion
    # of the testbench and writes it to a different output vector
    # file

    @decode="";
```

```
   @addr="";
   push(@decode,"0000000000");
   @addr=split(//,$tokens[4]);

   # Decode 4 character hex encoded data value to a 16 bit binary
   # encoded data value and push it into an array

   for ($i=0;$i<4;$i++)
   {
    $dummy=shift(@addr);

        if($dummy eq "0")
       {
push(@decode,"0000");
   }
   if($dummy eq "1")
   {
push(@decode,"0001");
   }
   if($dummy eq "2")
   {
push(@decode,"0010");
   }
   if($dummy eq "3")
   {
push(@decode,"0011");
   }
   if($dummy eq "4")
   {
push(@decode,"0100");
   }
   if($dummy eq "5")
   {
push(@decode,"0101");
   }
   if($dummy eq "6")
   {
push(@decode,"0110");
   }
   if($dummy eq "7")
   {
push(@decode,"0111");
   }
   if($dummy eq "8")
   {
       push(@decode,"1000");
   }
   if($dummy eq "9")
```

```
    {
        push(@decode,"1001");
    }
    if($dummy eq "A")
    {
        push(@decode,"1010");
    }
    if($dummy eq "B")
    {
        push(@decode,"1011");
    }
    if($dummy eq "C")
    {
        push(@decode,"1100");
    }
    if($dummy eq "D")
    {
        push(@decode,"1101");
    }
    if($dummy eq "E")
    {
        push(@decode,"1110");
    }
    if($dummy eq "F")
    {
        push(@decode,"1111");
}
    }

    push(@decode,"0");
    push(@decode,$tokens[5]);
    push(@decode,$tokens[6]);

    # Write the decoded vector array into a file

    for ($z=0;$z<$#decode+1;$z++)
    {
       print(OUTPUT1 $decode[$z]);
    }
  }
```

The following Perl script parses the input and output test vector files generated by the emulator into a single file in the RTL simulation testbench format so that the comparison utility provided by Texas Instruments can be used to check for errors.

```
# Perl script to parse the emulator input and output and generate
# vectors in the standard RTL simulation testbench format for RS coder

open(EMUINPUT,"write_data");     # Input vector file
open(EMUOUTPUT,"read_data");     # Input vector file
open(OUTPUT,">emu.vectors");     # Output file to store testbench

# Clock information is not returned by the emulator. The corresponding
# column in the testbench will be shown as 'X'

print "Note that clock information is not included.....!!!!!\n";

# This routine reads test vectors from the input test vector file
# and the output result vector file, combines them and writes the
# combined vector into the output file

while(!eof(EMUOUTPUT))
  {
# Split the lines using space as the delimiter
    @input_line1=split(//,<EMUINPUT>);
    @input_line2=split(//,<EMUINPUT>);

    @output_line=split(//,<EMUOUTPUT>);

    @ipline="";

# Store X for the clock bit
    push(@ipline,"X");
# Reset signal value
    push(@ipline,$input_line2[28]);
    push(@ipline," ");

# Encode 11 bit binary address into a 3 character hex address
# Call bin2dec to convert binary to decimal and print out the
# decimal value as hex to the file using %X
    for ($i=0;$i<3;$i++)
    {
if ($i==0)
  {
    $temp="";
    $temp=join '',$input_line2[29],$input_line2[30],$input_line2[31];
    @_=$temp;
    $dec=&bin2dec;
    push(@ipline,(sprintf "%X",$dec));
  }
if ($i==1)
  {
    $temp="";
    $temp=join '',,$input_line1[0],$input_line1[1],$input_line1[2],
                  $input_line1[3];
    @_=$temp;
    $dec=&bin2dec;
    push(@ipline,(sprintf "%X",$dec));
  }
if ($i==2)
  {
    $temp="";
    $temp=join '',$input_line1[4],$input_line1[5],$input_line1[6],
                  $input_line1[7];
    @_=$temp;
    $dec=&bin2dec;
    push(@ipline,(sprintf "%X",$dec));
  }
    }

# Store control signal values into the file

    push(@ipline," ");
    push(@ipline,$input_line1[8]);
    push(@ipline,$input_line1[9]);

    $temp="";
    $temp = join '',$input_line1[10],$input_line1[11],$input_line1[12],
            $input_line1[13],$input_line1[14];

    if ($temp eq "00001")
      {
push(@ipline,"0");
      }

    if ($temp == "11111")
      {
push(@ipline,"1");
      }

# Encode the 16 bit binary data value into a 4 character hex string
# Uses bin2dec to convert binary to decimal. The decimal value is
# printed to the file as hex using %X

    push(@ipline,$input_line1[15]);
    push(@ipline," ");

    for ($i=0;$i<4;$i++)
    {
if ($i==0)
  {
    $temp="";
    $temp=join '',$input_line1[16],$input_line1[17],
                  $input_line1[18],$input_line1[19];
    @_=$temp;
    $dec=&bin2dec;
    push(@ipline,(sprintf "%X",$dec));
  }
if ($i==1)
  {
    $temp="";
    $temp=join '',$input_line1[20],$input_line1[21],
                  $input_line1[22],$input_line1[23];
    @_=$temp;
    $dec=&bin2dec;
    push(@ipline,(sprintf "%X",$dec));
  }
if ($i==2)
  {
    $temp="";
    $temp=join '',$input_line1[24],$input_line1[25],
                  $input_line1[26],$input_line1[27];
    @_=$temp;
    $dec=&bin2dec;
    push(@ipline,(sprintf "%X",$dec));
  }
if ($i==3)
  {
    $temp="";
    $temp=join '',$input_line1[28],$input_line1[29],
                  $input_line1[30],$input_line1[31];
    @_=$temp;
    $dec=&bin2dec;
    push(@ipline,(sprintf "%X",$dec));
  }
    }

# Encode the 16 bit binary data value into a 4 character hex string
# Uses bin2dec to convert binary to decimal. The decimal value is
# printed to the file as hex using %X

    push(@ipline," ");

    for ($i=0;$i<4;$i++)
    {
if ($i==0)
  {
    $temp="";
    $temp=join '',$output_line[10],$output_line[11],
                  $output_line[12],$output_line[13];
    @_=$temp;
    $dec=&bin2dec;
    push(@ipline,(sprintf "%X",$dec));
  }
if ($i==1)
  {
    $temp="";
    $temp=join '',$output_line[14],$output_line[15],
                  $output_line[16],$output_line[17];
    @_=$temp;
    $dec=&bin2dec;
    push(@ipline,(sprintf "%X",$dec));
  }
if ($i==2)
  {
    $temp="";
    $temp=join '',$output_line[18],$output_line[19],
                  $output_line[20],$output_line[21];
    @_=$temp;
    $dec=&bin2dec;
    push(@ipline,(sprintf "%X",$dec));
  }
if ($i==3)
  {
    $temp="";
    $temp=join '',$output_line[22],$output_line[23],
                  $output_line[24],$output_line[25];
    @_=$temp;
    $dec=&bin2dec;
    push(@ipline,(sprintf "%X",$dec));
  }
    }

# Store control signals in the file

    push(@ipline," ");
    push(@ipline,$output_line[27]);
    push(@ipline," ");
    push(@ipline,$output_line[28]);
    push(@ipline,$output_line[29]);
    push(@ipline,$output_line[30]);
    push(@ipline,$output_line[31]);

# Write the vector array into the output file

    for ($z=0;$z<$#ipline+1;$z++)
    {
       print(OUTPUT $ipline[$z]);
    }
    print(OUTPUT "\n");

  }

# This subroutine converts a binary number into decimal

sub bin2dec {
       return unpack("N", pack("B32", substr("0" x 32 . shift, -32)));
   }
```

Appendix D

This appendix contains the RTL for the memory wrapper used for one of the Reed Solomon coder memories (described in Section 3.3.4). In this example, Mem1 is the memory which is used by the Reed Solomon coder. Mem2 is the memory available on the emulator.

```
// This is the memory used in the RS coder
module Mem1(A,CLK,D,EZ,WZ,Q);


input [6:0]   A;    // Input address port (read/write)
input         EZ,WZ,CLK;  // Enable signals and clock input
input [7:0]   D;    // Input data port (write)

output [7:0] Q;     // Output data port (read)
reg [7:0] Q;

// Mem2 signals to which values have to assigned depending on the
// values taken by the ports of Mem1

reg [6:0]  aadr;
reg [6:0]  badr;
reg [7:0]  bdata;
wire [7:0]  adata;
reg        ren,wen;

// Instantiation of Mem2, which is the memory available on the
// emulator. This memory has a separate read and write port
// and separate read and write enables

Mem2 Mem2
         (
          .ADATA(adata),  // Read address port
          .AADR(aadr),    // Read data port
          .REN(ren),      // Read enable
          .WEN(wen),      // Write enable
          .BDATA(bdata),  // Write data port
          .BADR(badr)     // Write address port
         );

// wen must be asserted when EZ and WZ are low and CLK is high
// write mode
always @(posedge CLK)
begin
   if ((!EZ)&&(!WZ))
      wen<=1'b1;
   else
      wen<=1'b0;
end

// ren must be asserted when EZ and WZ are not low and CLK is high
// read mode
always @(posedge CLK)
begin
   if ((!EZ)&&(!WZ))
      ren<=1'b0;
   else
      ren<=1'b1;
end

// When in write mode, badr must get the address value on A
always @(posedge CLK)
begin
   if ((!EZ)&&(!WZ))
      badr<=A;
end

// When in write mode, bdada must get the data value on D
always @(posedge CLK)
begin
   if ((!EZ)&&(!WZ))
      bdata<=D;
end

// If in write mode, aadr gets the value on A
always @(posedge CLK)
begin
   if ((!EZ)&&(WZ))
      aadr<=A;
end

// If writing, Q gets the value on bdata, else Q gets the value
// on adata
always @(posedge CLK)
begin
   if ((!EZ)&&(!WZ))
      Q<=bdata;

   if ((!EZ)&&(WZ))
```

```
      Q<=adata;
end

endmodule

// We need an empty module definition for Mem2 to keep the synthesis
// tool and the emulator compiler happy

module Mem2
           (
            ADATA,
            BDATA,
            WEN,
            REN,
            AADR,
            BADR
           );

   input [6:0] AADR,BADR;
   input [7:0] BDATA;
   input       REN,WEN;

   output[7:0] ADATA;

endmodule //
```

BIBLIOGRAPHY

[1] I. S. Reed and G. Solomon. "Polynomial Codes over Certain Finite Fields," *SIAM Journal of Applied Mathematics*, vol. 8, pp. 300–304, 1960.

[2] A. J. Viterbi. "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm," *IEEE Transactions on Information Theory*, April 1967.

[3] L. Semeria and A. Ghosh. "Methodology for Hardware/Software Co-Verification in C/C++," in *Asia and South Pacific Design Automation Conference*, January 2000.

[4] R. Tessier, J. Babb and A. Agarwal. "Virtual wires: Overcoming Pin Limitations in FPGA-Based Logic Emulators," in *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, April 1993, pp. 142–151.

[5] C. Selvidge, M. Kudlugi, S. Hassoun and D. Pryor. "A Transaction Based Unified Simulation/Emulation Architecture for Functional Verification," in *Proceedings of the 38th Design Automation Conference*, June 2001.

[6] C. Selvidge, M. Kudlugi and R. Tessier. "Static Scheduling of Multiple Asynchronous Domains for Functional Verification," in *Proceedings of the 38th Design Automation Conference*, June 2001.

[7] E. Barke, K .Harbich, J .Stohmann and L. Schwoerer. "A Case Study : Logic Emulation - Pitfalls and Solutions," in *Proceeding of the 10th IEEE International Workshop on Rapid System Prototyping*, June 1999, pp. 160–163.

[8] Ray Turner. "System Level Verification - A Comparison of Approaches," in *Proceedings of the 10th IEEE International Workshop on Rapid System Prototyping*, June 1999, pp. 154–159.

[9] Anant Agarwal. "VirtualWires - A Technology for Massive Multi-FPGA Systems", http://www.ikos.com

[10] K.Bartleson. "A New Standard for System Level Design," Synopsys Inc., http://www.systemc.org, September 1999.

[11] John Stickley and Duaine Pryor. "Functional Requirements Specification:Standard Co-Emulation Modeling Interface (SCE-MI)," Ikos Systems, February 2001.

[12] Scott Hauck. The Roles of FPGAs in Reprogrammable Systems. In *Proceedings of the IEEE, pp. 615-638*, Apr. 1998.

[13] Consultative Committee for Space Data Systems. "Recommendations for Space Data Systems Standard: Telemetry Channel Coding," Blue Book Issue 2, CCSDS 101.0-B2, January 1987.

[14] Vibhor Garg. "A PCI-X Bus RTL Transactor Model for System-on-a-Chip(SoC) Verification using Co-Modeling," M.S. Thesis, University of Massachusetts, Amherst, June 2001.

[15] Sriram Swaminathan. "An FPGA Based Adaptive Viterbi Decoder," M.S. Thesis, University of Massachusetts, Amherst, June 2001.

[16] S. B. Wicker. *Error Control Systems for Digital Communication and Storage*, Englewood Cliffs, N.J.: Prentice-Hall, 1994.

[17] Stephen B. Wicker and Vijay K. Bhargava. *Reed Solomon Codes and their Applications*, IEEE Press, 1994.

[18] E. R. Berlekamp. *Algebraic Coding Theory*, New York: McGraw-Hill, 1968.

[19] Ikos Systems Inc. *VirtuaLogic 3.5 User Guide*, March 2001.

[20] Texas Instruments Inc. "An Implementation of a Flexible Programmable Reed-Solomon Coder", March 1998.

[21] Mitch Dale. "Transaction Interface Portal," Ikos Systems Inc., May 2000.

[22] Cadence Design Systems Inc., http://www.cadence.com, *Transaction Based Verification.*

[23] Ikos Systems Inc., *Architectual Notes : TIP*, January 2000.

[24] Ikos Systems Inc., *TIP Co-Modeling Environment SRS*, February 2000.

[25] Ikos Systems Inc., *TIP User's Guide*, March 2001.

[26] *SystemC User's Guide, Version 1.2*, http://www.systemc.org, 2001.

[27] Ikos Systems Inc., http://www.ikos.com, *VirtuaLogic Asic Emulation Datasheet.*

[28] Cadence Design Systems Inc., http://www.cadence.com, *Verilog-XL Datasheet.*

[29] Quickturn Inc., http://www.quickturn.com, *Cobalt Datasheet.*

[30] Cadence Design Systems Inc., *NC-Sim Datasheet*, http://www.cadence.com.

[31] Synopsys Inc., http://www.synopsys.com, *Cyclone Datasheet.*

[32] Ikos Systems Inc., http://www.ikos.com, *NSIM Datasheet.*

[33] Mentor Graphics Corp., http://www.mentor.com, *Celaro Datasheet.*

[34] Synopsys Inc., http://www.synopsys.com, *Design Compiler Datasheet.*

[35] Synopsys Inc., http://www.synopsys.com, *Co-Centric SystemC Compiler Datasheet.*

[36] Quickturn Inc., http://www.quickturn.com, *System Realizer Datasheet.*

[37] Synopsys Inc., http://www.synopsys.com, *Hardware/Software Co-Verification with Synopsys Eagle Tools.*

[38] Mentor Graphics Corp., http://www.mentor.com, *HW/SW Co-Verification Technology.*