

Modeling and Synthesis with SystemC

Anup Varma

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

Dr. James R. Armstrong, Chair
Dr. F. Gail Gray
Dr. James M. Baker

December 2001
Blacksburg, Virginia

Keywords: SystemC, CoCentric, GSM, modeling, synthesis

Modeling and Synthesis with SystemC

Anup Varma

Abstract

With the increasing complexity of Application Specific Integrated Circuits (ASICs), System-On-a-Chip (SoC) design seems to be the current chip design paradigm. Unlike ASICs, SoCs are a potpourri of diverse components, including general-purpose or special-purpose processors. Designing and testing these designs require a new methodology that supports system level modeling and hardware-software co-design. The Hardware Description Languages (HDLs) available today cannot meet this challenge.

SystemC is a new modeling language based on C++. Models written in SystemC are executable and do not dictate either hardware or software implementation. The model written in SystemC can be synthesized to hardware using the CoCentric SystemC Compiler (CCSC). Thus, the combination of SystemC and CCSC has the potential to be a powerful SoC design technique.

This thesis examines the usefulness of SystemC and CCSC to model and synthesize a GSM system. The encoders and decoders used in the GSM system are complex and represent challenging problems in the real world. The modeling methodology using SystemC is considered and the synthesis issues with CCSC are detailed. Simulation results using real sound samples and synthesis results are presented. Areas for future work are then outlined.

Acknowledgements

I would like to thank Dr. James R. Armstrong for allowing me to work in this project. Always active, he would come up with innovative ideas to steer our work in the right direction. His enthusiasm was a source of inspiration to me. Working with him was a memorable experience.

I am grateful to Dr. James M. Baker for his support in this work and for serving as a member of my committee. I am thankful to Dr. F. Gail Gray for being on my masters' committee.

Installation and license issues are common for work involving software, and this one was no exception. Thanks to the patience and hard work of John Harris and Jos Sulisty, we were able to get around these quickly. I appreciate the work done by these friendly system administrators.

Many of my friends have helped me in different ways during the course of this work. While some of them provided help with the technical aspects of the work, others helped me with their ideas that improved my understanding of the scope of my work. I am thankful to all of my friends.

This work would not have been possible without the financial support from Motorola. I am grateful to them for this opportunity.

I would like to thank Synopsys for providing us with the CoCentric SystemC Compiler and also for their technical support.

Contents

Abstract	ii
Acknowledgements	iii
1 Introduction	1
1.1 Current Trends in Chip Design	1
1.2 SoC Design Issues	3
1.2.1 Hardware-Software Co-design	3
1.2.2 Modeling Tools	4
1.2.3 Design Synthesis	5
1.3 Aim of This Thesis	6
1.4 Task Description	6
1.5 Thesis Organization	7
2 The SystemC Language	8
2.1 Language Features	9
2.1.1 Modules, Ports and Signals	9
2.1.2 Processes	10
2.1.3 Data Types and Constructs	10
2.2 Modeling with SystemC	11
2.3 Other Languages	12
2.3.1 SpecC	12
2.3.2 Cynlib	12
2.3.3 Superlog	12
2.4 Reasons for Choosing SystemC	13
3 SystemC Synthesis	14
3.1 Synchronous Sequential Systems	14
3.2 Synthesis Tool Operation	15
3.3 CoCentric SystemC Compiler	17
3.4 Other Synthesis Tools	17
3.5 Reasons for Selecting CCSC	17
4 The GSM System	18
4.1 Speech Encoder	20
4.2 Parity Encoder	21
4.3 Convolutional Encoder	22
4.4 Interleaver Encoder	23
4.5 Packet Format Encoder	24
4.6 Encryption	25

4.7	Differential Encoder.....	27
4.8	Channel.....	27
4.9	Decoders.....	27
4.9.1	Viterbi Decoder.....	27
4.9.2	Speech Decoder.....	30
4.10	Reasons for Selecting GSM System.....	30
5	Previous Work.....	31
6	GSM System Modeling.....	32
6.1	Data Transfer Model.....	32
6.1.1	Bit-Parallel Transfer.....	32
6.1.2	Word Transfer.....	33
6.2	Communication Protocol.....	34
6.3	Module Implementation.....	36
6.3.1	Data Structures.....	37
6.3.2	Inter-process Communication.....	37
6.3.3	Separation of Communication and Computation.....	41
6.4	Process Implementation.....	41
6.4.1	Input Process.....	41
6.4.2	Data Process.....	42
6.4.3	Output Process.....	43
7	GSM Model Synthesis.....	45
7.1	Synthesis Tool Suite.....	45
7.2	The Synthesizable Subset of SystemC.....	45
7.2.1	Data Types.....	46
7.2.2	Language Constructs.....	46
7.3	Memory Mapping.....	46
7.4	Clock Frequency Selection.....	49
7.5	Scheduling Modes.....	50
7.5.1	Importance of wait () statements.....	50
7.5.2	Cycle Fixed Mode.....	51
7.5.3	Super-state Fixed Mode.....	52
7.6	BCView.....	53
7.6.1	HDL Browser.....	55
7.6.2	FSM Viewer.....	55
7.6.3	Reservation Table.....	56
7.6.4	Selection Inspector.....	57
7.6.5	Scheduling Error Analyzer.....	58
7.7	Memory and Tool Issues.....	59
8	Results.....	61

8.1	GSM Modeling.....	61
8.2	Synthesis Results.....	67
9	Conclusions and Future Work.....	69

List of Figures

Figure 1.1 A typical SoC.....	3
Figure 2.1 Modules, ports, signals and their relation.....	9
Figure 2.2 Sample SystemC model of a D Flip-flop showing the header file, dff.h (left) and the C++ file, dff.cpp (right). Code adapted from [10].....	11
Figure 3.1 Behavioral synthesis example. The code shown on the left side is mapped to the hardware circuit on the right	14
Figure 3.2 Flowchart showing the synthesis tool operation.....	16
Figure 4.1 GSM system showing the transmitter and receiver blocks.....	19
Figure 4.2 Block diagram of the RPE-LTP speech encoder. Figure adapted from [26]..	20
Figure 4.3 Block diagram of the parity encoder.....	22
Figure 4.4 Operation of the GSM convolutional encoder.....	23
Figure 4.5 Diagonal interleaver.....	24
Figure 4.6 Structure of a normal burst (speech data). Adapted from [31]	24
Figure 4.7 A5 encoder.....	26
Figure 4.8 Trellis diagram for the GSM Viterbi decoder. Adapted from [6].....	29
Figure 4.9 Block diagram of the RPE-LTP speech decoder. Figure taken from [26].....	30
Figure 6.1 Bit-parallel data transfer between two modules	33
Figure 6.2 Word data transfer between two modules.....	34
Figure 6.3 Block diagram showing the inter-module communication signals.....	36
Figure 6.4 PMG showing the inter-process communication signals.....	38
Figure 6.5 Flowchart showing the inter-process communication model	40
Figure 6.6 Pseudo-code for the input process	42
Figure 6.7 Pseudo-code for the data process.....	43
Figure 6.8 Pseudo-code for the output process	44
Figure 7.1 Main window for creating exploratory memory in memwrap.....	48
Figure 7.2 SystemC code (left) and the corresponding FSM representation (right).....	50
Figure 7.3 Cycle fixed scheduling mode. SystemC code (above) with the corresponding pre-synthesis simulation result (left) and the post-synthesis simulation result (right)	51
Figure 7.4 Super-state fixed scheduling mode. SystemC code (above) with the corresponding pre-synthesis simulation result (left) and the post-synthesis simulation result (right).....	53
Figure 7.5 BCView showing the different windows.....	54
Figure 7.6 HDL Browser window.....	55
Figure 7.7 FSM Viewer window.....	56
Figure 7.8 Reservation Table window	57
Figure 7.9 Selection Inspector window.....	58
Figure 7.10 BCView with Scheduling Error Analyzer	59
Figure 8.1 Transmitted and received sound samples for Male Speech.....	62
Figure 8.2 Frequency components in the transmitted sound (left) and the received sound (right) for Male Speech	63
Figure 8.3 Transmitted and received sound samples for Female Speech	63
Figure 8.4 Frequency components in the transmitted sound (left) and the received sound (right) for Female Speech.....	64
Figure 8.5 Transmitted and received sound samples for Music.....	64

Figure 8.6 Frequency components in the transmitted sound (left) and the received sound (right) for Music	65
Figure 8.7 Transmitted and received sound samples for Tone	65
Figure 8.8 Frequency components in the transmitted sound (left) and the received sound (right) for Tone.....	66

List of Tables

Table 6.1 Summary of the main control signals that implement the communication protocol.....	35
Table 6.2 Signals used in the inter-process communication model.....	38
Table 7.1 Comparison of array implementations.....	47
Table 7.2 Memory parameters used for synthesis.....	49
Table 8.1 Sound samples used for testing the system.....	61
Table 8.2 MSE for sound samples.....	66
Table 8.3 Summary of simulation results.....	67
Table 8.4 Synthesis results showing the timing, area and power for the hardware modules.....	68

1 Introduction

Integrated circuits (ICs) constitute the heart of most of the electronic devices available today. Typical systems use one or more ICs or ‘chip’s to respond to the ever increasing demand for high performance and low cost. There are several advantages of using chips compared to using discrete components like transistors in a circuit. Some of them are

- High circuit density

The functionality of large circuits with discrete components can be fabricated on a single chip. For instance, the Pentium 4 1.5GHz chip, measuring just 14mm X 16.05mm[1], replaces around 42 million transistors [2].

- High reliability

ICs are more rugged and reliable compared to discrete components. Since all the circuit units reside in one compact package, there is no chance of any physical breaks in interconnects.

- Low cost

Though the set up costs for IC plants are high, the chips are cheap when manufactured in large quantities.

Since most digital ICs replicate the functionality of circuits consisting of *at least* hundreds of transistors, designing the chips is not an easy task. Tools are required at many levels of the design process to simulate the circuit, verify the constraints, create the final layout, etc. Such software tools are commonly called Electronic Design Automation (EDA) tools.

1.1 Current Trends in Chip Design

Usually, the functionality of a chip is implemented in hardware. In such a scenario, the chip can be used only for the intended application and is termed an Application Specific Integrated Circuit (ASIC). Such chips cannot be re-programmed to perform another function. Most of the proprietary chips fall in this category, since it is very difficult to reverse engineer an ASIC to understand its operation. Field Programmable Gate Arrays

(FPGAs) can be re-configured, and these alter this scenario somewhat. A typical circuit consists of one or more ASICs and some discrete components on a circuit board.

Even though this methodology is sufficient for reasonably complex systems, a shift in design paradigm is required, as systems get more complex. Some of the reasons for this shift are

- Use of embedded software

If software is used in addition to the hardware, more complexity can be packed into the same chip. The functionality implemented in software can be easily re-configured by altering the program. Some functionality is inherently suited for software implementation. The software code can be stored in memory and executed by a general-purpose processor unit or a DSP unit.

- Design reuse

If a well-tested component is available, it is easier to use it in the design rather than developing it from scratch. There are many Intellectual Property (IP) vendors who sell commonly used component “core”s. Integrating these to build a chip reduces the chip development time.

This new design paradigm, incorporating the concepts of using diverse components, both software as well as IP cores, results in what is termed System-on-Chip (SoC). A SoC is usually much more complex than an ASIC and is the current trend in chip design.

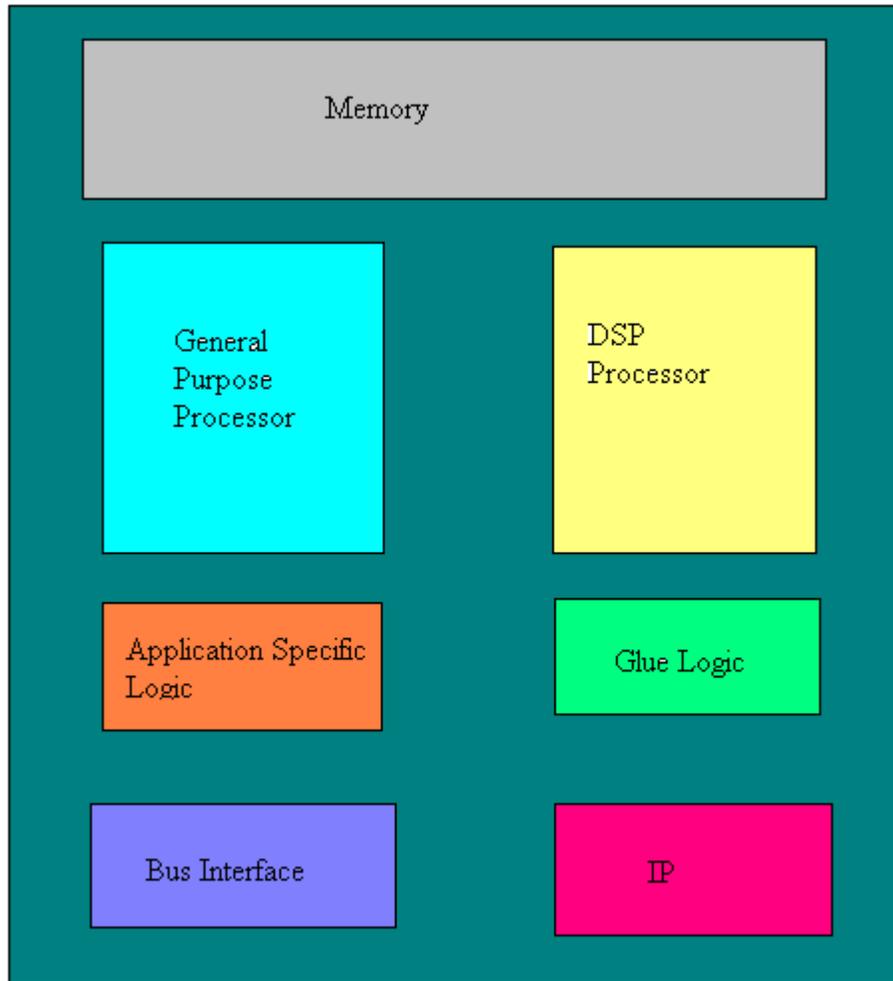


Figure 1.1 A typical SoC

1.2 SoC Design Issues

While SoC design has a promising future, certain issues have to be taken into account if this paradigm is to drive the chip design industry. Most of these challenges are unique to this style of design, and are active research areas.

1.2.1 Hardware-Software Co-design

In SoC design, a given specification must be broken down to two main categories – hardware and software. This problem is called the hardware-software partitioning problem and is an important issue that can affect the performance of the overall system.

The goal is to partition the components of the system to either hardware or software so that the resulting system provides optimum performance. Usually, the partitioning is done based on the evaluation of a cost function whose exact form depends on the design constraints [3]. For example, execution time, data memory, frequency of activation, etc. for a unit can be included in its cost function. This process is not standardized and a typical analysis can yield multiple partitions, making the design process an iterative one where each partition is selected and evaluated one by one until the design specifications are met at the lowest cost. Also, it is difficult to automate this process completely and user intervention is usually required. This problem is an active research area, and success has been achieved in some cases.

Another closely related issue is testing these heterogeneous components. While we can test hardware and software independently, it is important that the testing is done simultaneously, considering the following points.

- The interfaces between the hardware and software units should be tested.
- If the hardware part is tested first and synthesized, specification changes and bug fixes end up in software. This may adversely impact the performance.

Thus, a co-simulation environment is required for simultaneous hardware-software validation.

1.2.2 Modeling Tools

Hardware description languages (HDLs) are an important set of EDA tools. These are high level programming languages that are designed to model the behavior of hardware. Two of the most popular HDLs are the Very High Speed Integrated Circuit Hardware Description Language (VHDL) and Verilog. HDLs have semantics that model typical hardware features like concurrency, delay, clock, ports and signals. Once a model is coded in an HDL, it can be simulated to verify the intended functionality and then synthesized to hardware. While this approach is suited for creating chips that contain only hardware part, it is inappropriate for SoC design. This is because

- Typical systems can be represented at higher levels of abstraction than can be handled by HDLs. Languages such as C and C++ are commonly used to

express highly abstract systems. Because of higher simulation speed, functional verification is always preferred at the highest level of abstraction possible. Verification of C/C++ models is much faster than those written in HDLs.

- Current HDL simulation environments do not support hardware-software co-design and co-simulation.
- Well-tested IP in C/C++ for commonly used systems is available in the market and using this can drastically reduce the chip design time, instead of starting from scratch.

Because of the limitations of HDLs for designing SoCs, System Level modeling Languages (SLMLs) are used for this purpose. One of the recent developments in this field is the emergence of a new modeling language named SystemC [4]. This language is based on C++ and has specific constructs to model hardware-related concepts. Details of SystemC are covered in the next chapter.

1.2.3 Design Synthesis

Usually, the model for functional verification is written at a high level of abstraction, without directly employing any hardware components. This is done because while it is difficult to represent non-trivial systems at the circuit level, it is much easier to represent their behavior at an algorithmic level. Another reason is that simulation speed increases with the level of abstraction. Thus the main task after functional verification is the generation of the actual circuit that the model represents. There are tools available in the market to perform this task and these tools are generally termed synthesis tools, since their function is to synthesize the circuit. Synthesis tools use predefined library components and map algorithmic code to component instantiations. To enable this translation, the algorithmic code must be written in a format specified by the tool vendor. Hence, adoption of SLMLs to the SoC design process would be possible only if adequate synthesis tools are available that read in the behavioral code and map it to hardware components, and the coding style requirements are followed.

A new development in the synthesis market is the introduction of the Synopsys CoCentric SystemC Compiler (CCSC) [5]. This tool reads code written in SystemC and

translates the design to hardware, using library components. The combination of SystemC and CCSC could be a very powerful methodology for designing complex chips in a short time window.

1.3 Aim of This Thesis

The goal of this thesis is to study the effectiveness of SystemC in modeling large, real-life systems, and evaluating the use of CCSC in synthesizing the system. While it is easy to test SystemC on simple designs, real-life designs are often big, and the performance of the language in modeling these designs decide whether it is accepted by the industry. Since modeling and synthesis are closely related, we study the usefulness of CCSC to synthesize our design.

1.4 Task Description

This thesis describes the work done in modeling the GSM system in SystemC and synthesizing it using CCSC. This effort was part of a larger work sponsored by Motorola. The following tasks were proposed for the sponsored work.

1. Model the GSM system at a high level of abstraction.
2. Refine the model to consider the data transfer based on words.
3. Synthesize the system to get the hardware parameters – timing, area and power
4. Execute the model on a StarCore DSP board and evaluate the timing.
5. With the hardware and software metrics obtained in the previous steps, run the partition algorithm to partition the system
6. Estimate the communication delays between the modules due to bus read/write for the hardware modules and software task switching delays for the software modules.
7. Back annotate the delay values to the hardware and software modules and verify the functionality.

In this thesis, we focus on modeling and synthesis. Specifically, we performed the steps 1, 2, and 3, and participated in steps 6 and 7.

Considerable work has been done in modeling the GSM system using VHDL at Virginia Tech [6]. We used this knowledge base to model our system. Also, modeling the speech

codec used in the GSM system from scratch is a complicated task, and hence we used the C code available in the public domain [7] as our template.

In addition to the modules in the GSM system, we needed to model some new units for simulation. For testing our model, we needed to read and write sound files in au format. Hence we required file reader and writer modules. Each encoder /decoder module works on data frames of different lengths, and data transfer through the channel is done bit-by-bit, in a serial fashion. Serializing/de-serializing modules had to be written to interface the modules with the channel. We also modeled an abstract digital channel that introduces random bit errors and burst errors. The user can control the probability of bit error and the length of error bursts.

1.5 Thesis Organization

Chapter 1 is the introduction to this thesis, and provides a summary of the background of this thesis. The task description and the thesis organization are also provided.

Chapter 2 describes the SystemC language. The features of the language are explained. Alternatives to SystemC are considered, and the reasons for selecting SystemC for this work are summarized.

Chapter 3 provides an overview of the synthesis process. It covers the details of CCSC.

A description of the GSM system is given in chapter 4. The functional units of the GSM system are explained.

Chapter 5 examines the current literature in the fields of GSM modeling, SystemC and synthesis. The motivation for this work is also presented.

Chapter 6 discusses the modeling methodology we followed in our work. The communication protocol between different modules of the system is explained, and the implementation of each module is described.

The work done in synthesizing the system using CCSC is detailed in chapter 7. It also examined the issues we faced during the process.

Chapter 8 provides the results of our modeling and synthesis work.

Chapter 9 states the conclusions drawn from our work and suggests possible directions for future research.

2 The SystemC Language

SystemC is a modeling language based on C++. It is a set of class libraries in C++ that allow users to model hardware related concepts like concurrency, timing etc. Thus, SystemC is an extension of C++ with classes to add the desired functions, thus using the object-oriented programming paradigm of C++. Since this language is basically C++, any ANSI C++ compiler can be used to run SystemC models. Because of these features, the language offers the following advantages.

- Executable Specification – A model written in SystemC can be compiled and made executable. This feature implies that the executable specification can be shared among the members of a team without the need to share the source code or any associated script file.
- Faster simulation – Since SystemC depends on the underlying C++ framework for simulation, the simulation speed is high compared to either VHDL or Verilog. In an experiment done at Virginia Tech [8], it was found that the ratio of the simulation speeds of SystemC and VHDL was at least 10:1 for the same model. The speedup could be higher with the use of commercial tools.
- Higher abstraction levels – Compared to hardware description languages, C++ has the ability to model highly abstract concepts in an elegant fashion. This feature is thus inherent in SystemC.
- Implementation independence – A model specified in a hardware description language is usually targeted for a hardware implementation. In contrast, a SystemC model does not specify a particular implementation. It can be implemented either in hardware or in software, using a general-purpose processor or a DSP. This feature is very useful in hardware-software co-design, as explained before.

Another important feature of SystemC is that it is provided as an open source distribution, where users can download the source code of the SystemC language. Hence the users can submit their bug fixes or add their own features to the language. This

ensures that the language evolves rapidly through the collective effort of the design community.

2.1 Language Features

The important features of SystemC that are useful for modeling are briefly discussed here.

2.1.1 Modules, Ports and Signals

The basic block of a SystemC program is a *module*. A module is similar to the concept of entity in VHDL and module in Verilog. It is an abstract representation of a functional unit, without specifying any implementation details. Each non-trivial module has a set of *ports* through which it interacts with the outside world. Ports can be input ports, output ports, or input/output (I/O) ports. If a module reads data, it must have at least one input or I/O port, and if it writes data, it should have one or more output or I/O ports. Individual modules communicate with one another through *signals* that connect the ports of the modules.

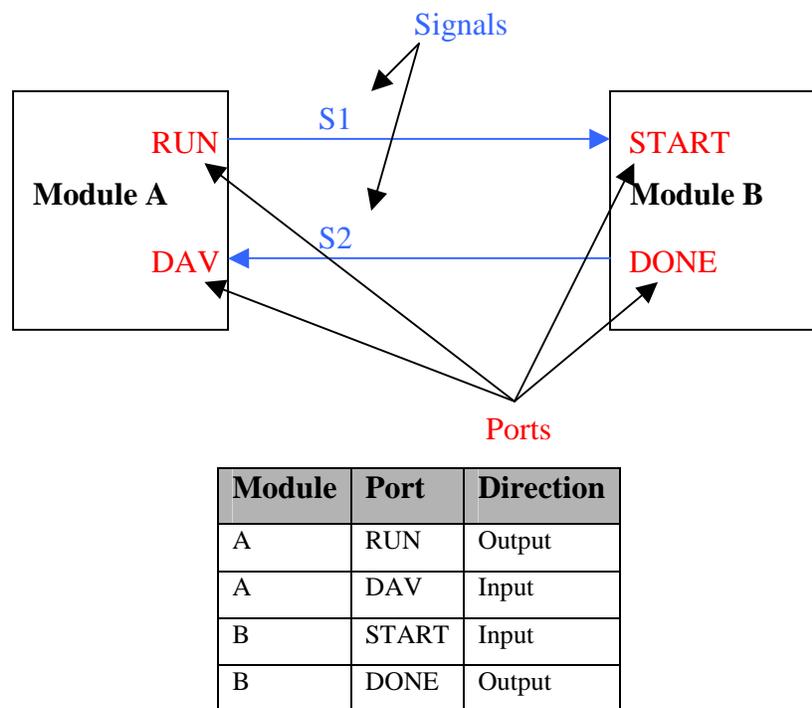


Figure 2.1 Modules, ports, signals and their relation

Thus, signals are similar to the wires that interconnect different hardware units on a circuit board, and the ports correspond to the pins of these units.

2.1.2 Processes

The code that implements the algorithm of a module is encapsulated in one or more *processes*. There are three types of processes in SystemC. This classification is based on how the underlying SystemC simulation kernel calls and executes the processes.

- Method Process (SC_METHOD) – Each method process has a *sensitivity list* that lists the signals that can activate this process. Whenever there is a change in one of the signals in this list, the process is executed. Once the process starts execution, it cannot be suspended and it runs until it returns.
- Thread Process (SC_THREAD) – This process is similar to SC_METHOD in that it also has a sensitivity list to control its activation. It can be suspended and re-activated by the user by adding relevant language constructs in the code. When the process is suspended, it waits until one of the signals in its sensitivity list changes. It then resumes execution from the point where it was suspended.
- Clocked Thread Process (SC_CTHREAD) – This process is a special case of SC_THREAD where the sensitivity list has only one signal, and it is activated when a specific edge (low to high or high to low) of that signal occurs. This activation scheme allows the modeling of synchronous designs in a simple manner.

2.1.3 Data Types and Constructs

SystemC is C++ with additional classes, and hence all C++ data types are supported. For modeling hardware, additional data types are available. These include types for representing bits, bit vectors, 4-valued logic, variable precision integers, etc.

In addition to these data types, the language also provides constructs that enable represent hardware behavior. There are wait () statements that suspend execution, write () and read () functions to send and receive data from ports, and so on. More details of the language

are discussed in [9], and a complete list of features can be found in the SystemC user's guide [10].

The language features can be illustrated with an example. Typically, the code is split to two files – a header file that describes the ports and the processes, and a C++ file that provides the implementation.

dff.h	dff.cpp
<pre>#include <systemc.h> SC_MODULE(dff) { sc_in<bool> din; sc_in<bool> clock; sc_out<bool> dout; SC_CTOR(dff) { SC_METHOD(action); sensitive_pos << clock; } };</pre>	<pre>#include "dff.h" void dff::action() { dout = din; }</pre>

Figure 2.2 Sample SystemC model of a D Flip-flop showing the header file, dff.h (left) and the C++ file, dff.cpp (right). Code adapted from [10]

The example in Figure 2.2 represents a D flip-flop. The code is split to two files – dff.h and dff.cpp. The header file, dff.h defines a module named dff. The module has an SC_METHOD process named *action* and it is declared to be sensitive to the positive edge of the signal *clock*. Every time the clock signal makes a low to high transition, the process is invoked. The implementation of the process *action* is given in the file dff.cpp.

2.2 Modeling with SystemC

The features of SystemC allow one to model a system at a high level of abstraction. The modeling process is an iterative one, with data and control refinement occurring at each step. Data refinement involves improvements in the way data types are modeled, and control refinement refers to the evolution of the control protocols in the model. The

refinements can be evaluated by performing simulations at each iteration. Once the refined model is finalized, it can be partitioned and synthesized.

2.3 Other Languages

There are other languages that are similar to SystemC in that they allow modeling hardware at a high level of abstraction. We briefly discuss some of these in this section. This list is by no means exhaustive.

2.3.1 SpecC

SpecC is a system specification description language based on C [11]. It is developed as a language extension to ANSI C, with special constructs to model hardware. A model can be described at a high level of abstraction in SpecC. It can then be simulated using the SpecC reference compiler (SCRC). SCRC consists of an open source pre-processor that converts the SpecC specific constructs to equivalent ANSI C code, which is then compiled and simulated [12], [13]. Thus, we get executable specifications of models. The main difference of SpecC from SystemC is that the modeling domain of SpecC covers higher levels of abstraction than that done using SystemC [14].

2.3.2 Cynlib

Another C++ based language for hardware modeling is Cynlib [15]. The focus of this language is hardware design, and not system level specification. At the time of writing this document, it is planned to merge Cynlib with the next version of SystemC to create a faster version of SystemC [16]. The Extended SystemC Cynlib (ESC) would have the simulation kernel of Cynlib to provide faster simulation, and is planned for release by December 2001 [17], [18].

2.3.3 Superlog

Superlog is a system level description language based on Verilog and C [19]. It is a superset of Verilog with C language-like constructs. Code written in Superlog cannot be compiled to executable format, and requires the availability of specific simulator tools [20].

2.4 Reasons for Choosing SystemC

We chose SystemC for our modeling task after considering many factors. SystemC is a new language that enjoys the support of many EDA vendors [21]. The language does not need any simulator, and it requires only an ANSI C++ compiler for simulation. The syntax is easy to learn, and we also have a graduate level course at Virginia Tech that covers the basics of the language [22]. Availability of synthesis tools was another important consideration [23], [5]. These factors naturally led us to adopt SystemC for this thesis work.

3 SystemC Synthesis

A synthesis tool reads the behavioral description of a model and translates that to an equivalent netlist. The tool has a set of hardware building blocks with well-defined parameters. The parameters include timing, area, power dissipation information, etc. When the tool parses the behavioral code, it maps the statements to the appropriate hardware components. This is shown by a simple example.

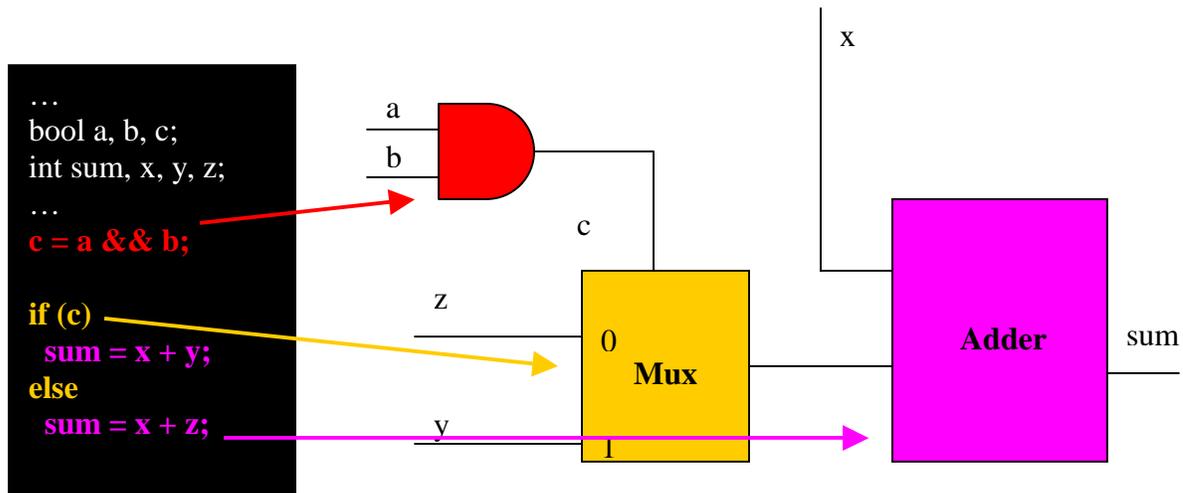


Figure 3.1 Behavioral synthesis example. The code shown on the left side is mapped to the hardware circuit on the right

The above example shows a simple case. In general, the tool infers hardware based on the coding style used in the model. The same model can be written in different ways without affecting the simulation results, but the synthesized circuit may be different in each case. This is because the tool is basically a piece of software that performs the hardware mapping based on a pre-defined algorithm. Hence care has to be taken to follow the correct coding style that is suited for a given scenario.

3.1 Synchronous Sequential Systems

While the above task of mapping software code to hardware library components is ideal for simple combinational circuits, additional considerations have to be taken to synthesize

synchronous sequential circuits. These circuits have at least one clock signal that controls the operation of the circuit. Typically, each clock cycle causes specific sub-units of the circuit to execute. Hence the hardware implementation of such a model consists of

1. Hardware components to perform the actions at each clocked state, and
2. Control unit to control the operation of the state machine, to ensure that the components in step 1 are activated at the appropriate clock cycles.

While the first point can be addressed by mapping the relevant lines of code to hardware, the second issue is more complicated, and is termed *scheduling*. A given set of hardware units can be scheduled in different ways to perform trade-off analysis.

3.2 Synthesis Tool Operation

In general, a synthesis tool performs at least the two tasks mentioned in the previous section. The general operation flow for a synthesis tool is given in Figure 3.2.

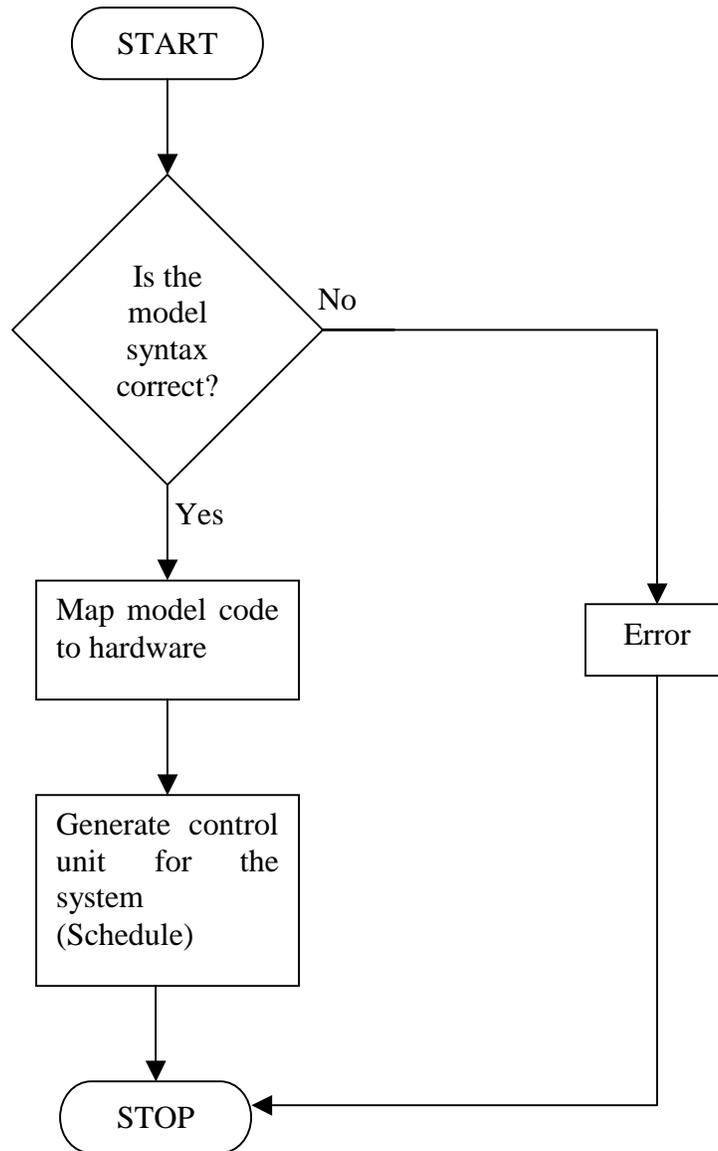


Figure 3.2 Flowchart showing the synthesis tool operation

The synthesis tool first reads the model and performs a syntax check to ensure that only synthesizable constructs are used in the model. If this check fails, the user is notified, and it is the task of the user to correct the code and run the tool again. After this, the tool maps the code to hardware components. The next step is scheduling the design to synthesize the control unit for the system. The result of this step is the netlist of the

synthesized design. This is an equivalent hardware representation of the system modeled at the behavioral level.

3.3 CoCentric SystemC Compiler

Synthesis from SystemC is a relatively new area, and hence the availability of synthesis tools that support SystemC is limited. One of the SystemC synthesis tool suites is the CoCentric SystemC Compiler and the design compiler (DC). CCSC reads a model written in SystemC and allocates hardware, and DC performs the scheduling operation. In addition, there is another tool, BCView that provides a graphical user environment for analysis after the scheduling step. Using this feature, the user can visualize the hardware allocation and utilization for each clock cycle, in addition to getting further information about the model. Some types of scheduling errors can be analyzed using this tool.

3.4 Other Synthesis Tools

Among the other tools that support SystemC, the one that is prominent is N2C. It reads a specification written in SystemC and generates synthesizable HDL code for the hardware part. There are several tools that synthesize hardware from HDL code. Hence N2C can be considered a SystemC synthesis tool. It also has provisions for hardware-software co-design and interface synthesis.

3.5 Reasons for Selecting CCSC

One of the main advantages of CCSC is that the final result is a netlist, like that obtained by synthesizing HDL code. This means that only one synthesis step, directly using SystemC is required. Another factor was the availability of the tool. Considering these, we decided to use CCSC for this work.

4 The GSM System

Global System for Mobile Communications (GSM) is the most popular cellular system in the world today [24]. It was developed to provide a standard for reliable, secure and flexible mobile communications without geographic restrictions. The system is digital and has a set of encoders and decoders to provide speech compression, error detection, error correction and encryption. A block diagram of the system showing the functional units is given in Figure 4.1.

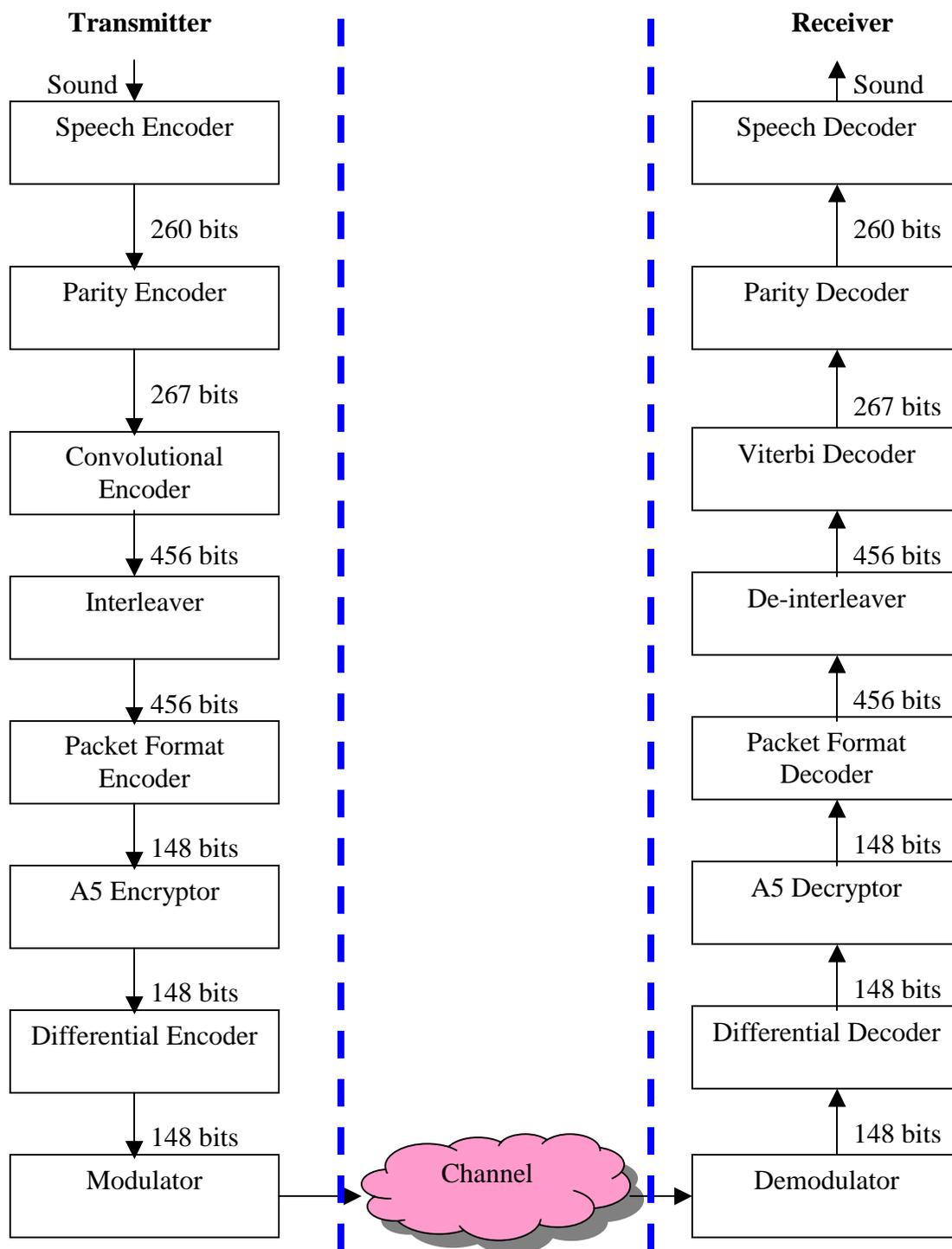


Figure 4.1 GSM system showing the transmitter and receiver blocks

In this thesis, we concentrate on the digital part of the GSM system, leaving out the modulator and demodulator blocks. Also, we replace the noisy analog channel with a digital model where the user can control the errors introduced by the channel. Brief descriptions of the encoders and the decoders modeled here are presented below. More details of each of these can be found in digital communications books [25].

4.1 Speech Encoder

The speech encoder converts human speech to data bits for transmission. It reads frames of 160 signed 13-bit PCM values sampled at 8 kHz and compresses it to frames of 260 bits each. One input frame of 160 values carries 20ms of speech. Hence the speech encoder gives out data at the rate of 13kbps (260bits / 20ms).

The encoder used in the GSM system is the Regular Pulse Excitation Long Term Predictor (RPE-LTP). Rather than compressing the speech data in a straightforward manner, this encoder models the human vocal tract and generates the parameters that are necessary to re-create the sound at the receiver.

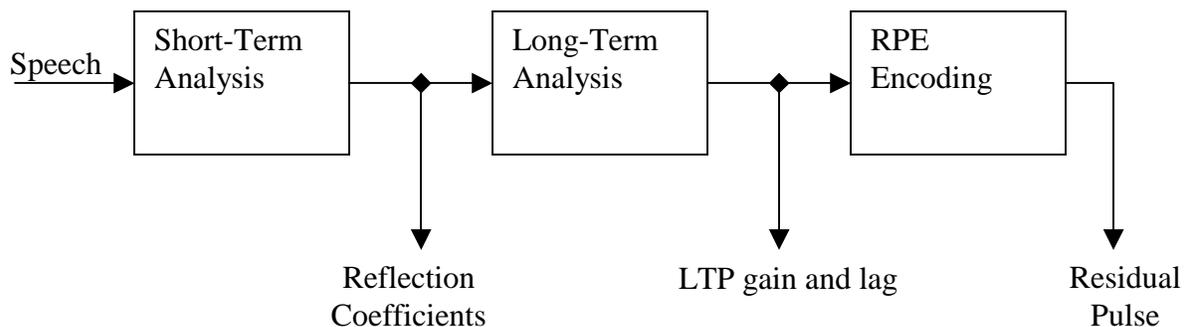


Figure 4.2 Block diagram of the RPE-LTP speech encoder. Figure adapted from [26].

The encoder models the human vocal system as two digital filters and an excitation sequence that, when passed through the filters, generates speech. The encoder consists of 3 blocks – the Short-Term Analysis section, the Long-Term Analysis section, and the RPE Encoder.

Incoming speech is analyzed by the Short-Term Analysis stage. The current speech sample can be approximated as the weighted sum of the last 8 samples. These weights are called Linear Prediction Coefficients (LPCs) and are computed by the Short-Term Analysis stage. When sound passes through two tubes of varying diameters, some part of the sound is reflected back at the junction of the tubes. This reflection is quantified by the parameter *reflection coefficients*. The reflection coefficients corresponding to the LPCs are calculated and transmitted as one set of parameters.

The Long-Term Analysis section reads 40 sound samples at a time, reconstructed from the short term analysis parameters. It computes the parameters LTP lag and LTP gain. The LTP lag corresponds to the inverse of the pitch of the speech and the LTP gain represents the scaling factor.

The RPE Encoder extracts the excitation sequence that is used to excite the filters in the decoder. This sequence is usually weak or random and can be easily compressed and transmitted. The encoded excitation sequence is then transmitted.

Details regarding the theory and performance of the RPE-LTP encoder can be found in [26], [27], [28], [29], [30].

4.2 Parity Encoder

This encoder provides error detection capability to the system by adding parity bits to the incoming bit stream. The parity encoder reads a frame of 260 bits. These 260 bits are divided into 50 Class 1A bits, 132 class 1B bits and 78 class 2 bits, based on their relative importance to the sound quality. Class 1A bits have the most impact on quality, followed by class 1b bits. Class 2 bits are the least important bits of the data stream. Hence, class 1 A bits are protected by both error detection and error correction coding, class 1B bits are covered by error correction coding and class 2 bits are not given any coding at all. Also, if any error is detected in the class 1A bits, the whole frame is rejected.

The parity encoder adds 3 parity bits to the class 1A bits and pads 4 '0's to class 1B bits. The zero padding is used to help the error correction decoder (Viterbi decoder) to correctly identify the transmitted bits. Thus the input of this encoder is 260 bits and the output is 267 bits.

The implementation of the parity encoder is done using 3 registers for parity bit generation.

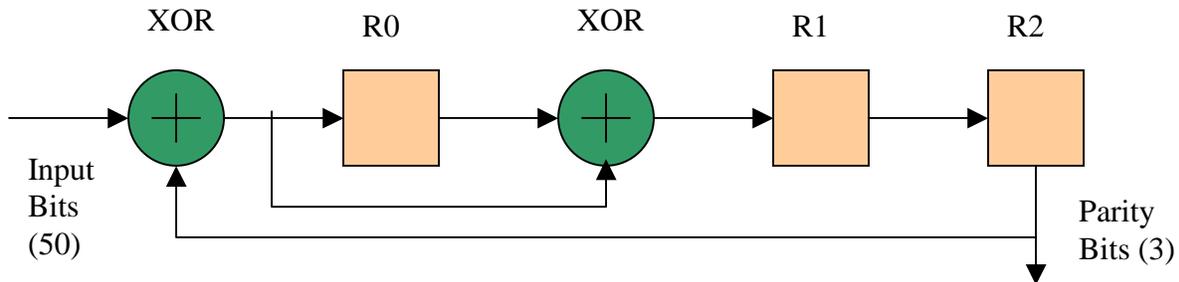


Figure 4.3 Block diagram of the parity encoder

After the incoming class 1A bits are shifted in, the system is clocked thrice to get the three parity bits, which are then padded to the bit stream. The parity decoder at the receiving end used the same set of LFSR to regenerate the parity bits. If the re-computed parity bits do not match the transmitted parity bits, there is an error and the whole frame is discarded.

4.3 Convolutional Encoder

The function of the convolutional encoder is to add redundant bits to the bit stream for error correction. In general, the input bits are stored in K k -bit shift registers connected in series, and the shift registers are clocked together. From these $K*k$ bits, n output bits are generated by selecting specific bits and performing modulo-2 addition. The rate, R , of the encoder is defined as the ratio k/n . A high value of n implies low value for R and high redundancy in the output bits, and thus more error correction capability. Hence as n increases, the error correction capability of the encoder increases. The disadvantage of having an arbitrarily large n is that the number of output bits that carry no useful information also increases, wasting bandwidth. Another important factor in the error correction capability is the way in which the selection of bits is made to get the n output bits. This point is discussed later when we consider the Viterbi decoder. A good discussion of convolutional encoders is given in [25].

The convolutional encoder used in the GSM system has the parameters $k = 1$, $n = 2$ and $K = 5$. It encodes the class 1A and class 1B bits in the incoming bit stream. There are 189 class 1 bits, and hence the encoder generates 378 coded bits. The 78 class B bits are left untouched. Hence the input to this encoder is a frame of length 267 bits and the output is 456 bits long.

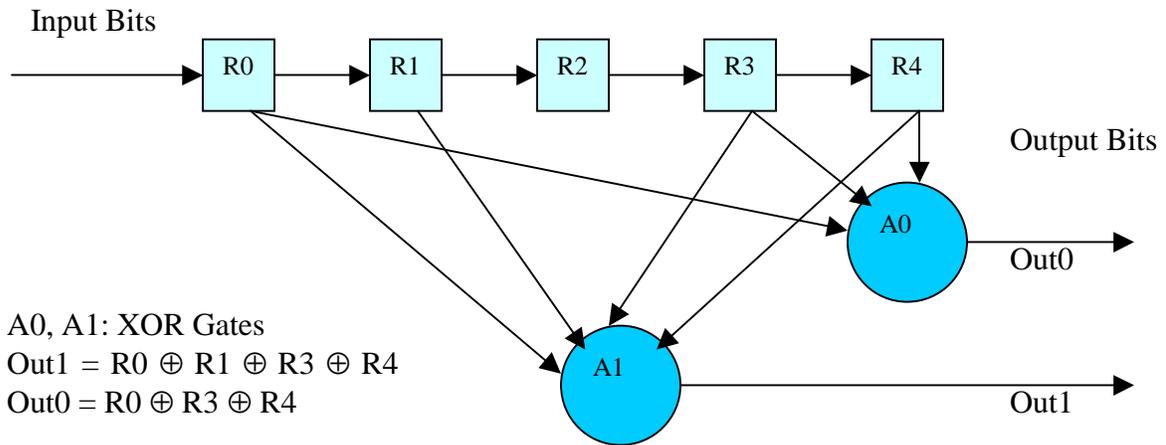


Figure 4.4 Operation of the GSM convolutional encoder

4.4 Interleaver Encoder

Channels in a wireless system usually introduce errors in bursts. Even strong error correction coding fails when the number of consecutive bit errors is large. An interleaver encoder alleviates this problem by re-ordering the transmitted bits in such a way that consecutive bit errors introduced by the channel are spread out in the received signal in a non-consecutive fashion.

The interleaver used in the GSM system is a diagonal interleaver. Its operation is pictorially described in Figure 4.5.

sub block	1	2	3	4	5	6	7	8
even bits	B0	B1	B2	B3	C0	C1	C2	C3
odd bits	A4	A5	A6	A7	B4	B5	B6	B7

Figure 4.5 Diagonal interleaver

In the above diagram, A, B, and C are three consecutive data frames, each having 456 data bits. They are combined and divided to 8 sub-blocks of 114 bits each, as shown in the diagram. Each sub-block has bits from 2 frames, and the bits from each frame are interleaved with those of the other frame so that no two bits from the same frame occur consecutively. This form of interleaving efficiently separates consecutive bits from the input stream.

4.5 Packet Format Encoder

Each data frame is further formatted to include control bits. There are several types of frames that can be sent, in addition to the speech data frame, and the formatting is different for each of them. In this thesis, we consider only the speech data frame, which is called a *normal burst*.



Figure 4.6 Structure of a normal burst (speech data). Adapted from [31]

In Figure 4.6, the upper text represents the type of bits and the lower text indicates the number of bits. The different types of bits are

- Tail bits (T) – These 3 bits at the start and end of a normal burst are used to separate bursts, and are set to zero.

- Data – Two sets of 57 bits of data from the interleaver are packed in a normal burst.
- Stealing Flags (S) – These bits indicate whether the burst carries signaling data or user data.
- Training Sequence – This fixed bit sequence of 26 bits is known to both the transmitter and the receiver in advance and is used for synchronization in environments where there is multipath fading.
- Guard bits – These 8.25 bits are not physical bits, but represent the time delay at the end of the burst when power control is used by the system. No bits are transmitted during this time.

4.6 Encryption

The data bits of the formatted frame are encrypted to enhance the privacy of communications. The algorithm used in the GSM system is known as the A5 algorithm and has not been officially published. This algorithm is available in the public domain at several Internet sites [32], [33] and also in books on encryption [34], but each description is slightly different, though they do have several common features. Our work uses a combination of [29] and [31]. However, since we are not testing the system for immunity from eavesdropping, the actual algorithm is not of much importance. The point here is that we are adding additional complexity to the system by introducing another computational unit.

Our version of the A5 algorithm uses 3 Linear Feedback Shift Registers (LFSRs) of lengths 19, 22 and 23 bits. Each LFSR is clocked independently, based on the value of its middle bit and that of the middle bits of the other registers. The clocking algorithm has the property that at any given time, two of the 3 registers are clocked. The operation of the module is shown in Figure 4.7.

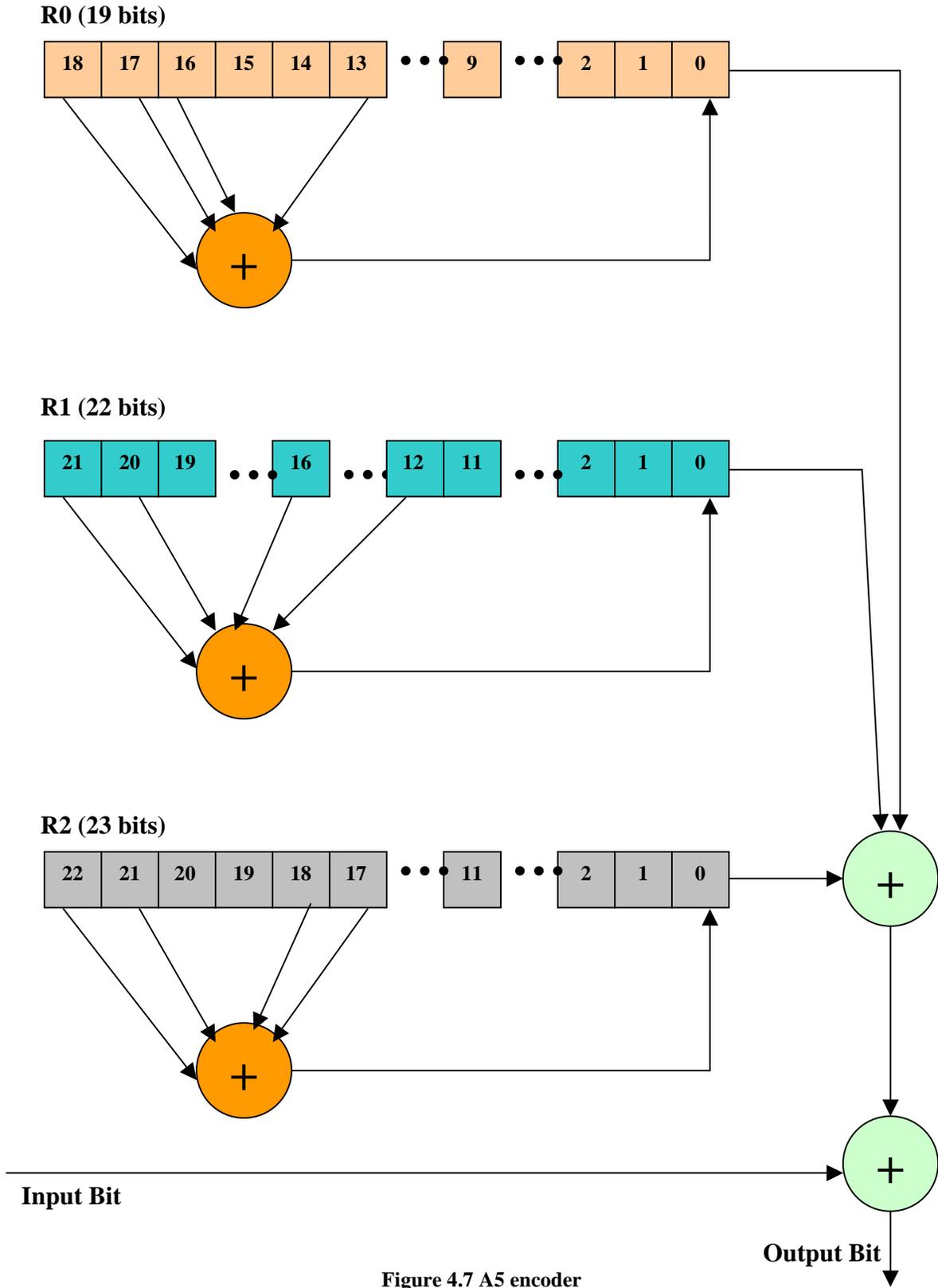


Figure 4.7 A5 encoder

The registers are initialized with a key that is then distributed in the system by clocking the registers 100 times. This is done to preserve the pseudo-random nature of the output bits. The decoder also uses the same key and performs the same algorithm to decode the sequence.

4.7 Differential Encoder

The modulation scheme used in the GSM system requires that the phase information of the data be known at the receiver, and to aid this process, differential encoding is used. This encoder gives an output bit depending on the values of the current bit and the previous bit. Basically, the encoder performs an XNOR operation of these bits.

4.8 Channel

The bits from the differential encoder are used for modulation and the modulated data is transmitted over a noisy channel. For the purpose of modeling this effect, we used a bit serializer that converts the frames to individual bits and a digital channel model.

The channel model can selectively introduce errors, either in single bits or in bursts of consecutive bits. Given a value for the probability of bit error, the digital channel model can randomly introduce errors to match the probability.

4.9 Decoders

At the receiver, each of the encoders discussed above has a corresponding decoder whose function is to reverse the processing of the encoder. Two of the decoders that deserve discussion are the Viterbi decoder and the speech decoder.

4.9.1 Viterbi Decoder

This decoder is used to retrieve the bits that are encoded by the convolutional encoder. The main feature of this decoder is its ability to correct errors. It was mentioned in the discussion of convolutional encoder that the way the n output bits of the convolutional encoder are selected affects the error correction capability of the system. This is because the Viterbi decoder uses an algorithm known as the Maximum Likelihood Sequence Estimation (MLSE).

The goal of MLSE is to find the best estimate of the transmitted bits from a set of received bits. The received bits may be in error. The estimation of the transmitted bits is based on the principle that only a small set of allowable symbols are transmitted, and if the received pattern does not match any of these exactly, the closest match is assumed to be the transmitted bit sequence.

For example, if a system transmits only the bits 0000, 0100, 1000, and 1111, and if the received bit sequence is 0101, it can be inferred that the received sequence is in error, since the receiver knows the set of valid transmission sequences. The goal is to make an intelligent estimate of the transmitted sequence, given the received sequence. The logical estimate would be the transmission sequence that is closest to the received sequence. Measures of 'closeness' can be done in different ways, and one of the commonly used one is the number of differences in the corresponding bits of the sequences. This measure is termed the Hamming distance. In this example, the minimum value of the Hamming distance between the received sequence and any of the transmitted sequences is 1, for the transmission stream 0100. The receiver then estimates that the bit sequence 0100 was transmitted. If the bit error rate is low, this estimate is likely to be true, and the system performs error correction. It is important that the transmitted sequences be as far apart from one another as possible, to reduce the possibility of incorrect estimation.

The convolutional encoder can be represented as a Finite State Machine with each input bit causing a state transition. The convolutional encoder used in the GSM system uses the current input bit and 4 previous bits to generate the output bits at any given time. Hence it has 16 states, and each incoming bit causes the unit to transition from one state to another, and each state has a 2-bit output. The Viterbi decoder uses the same state machine and its operation can be visualized using trellis diagrams. These are state diagrams that unfold in time, indicating all the possible sets of transmitted and received bits and the decoder state at each time interval.

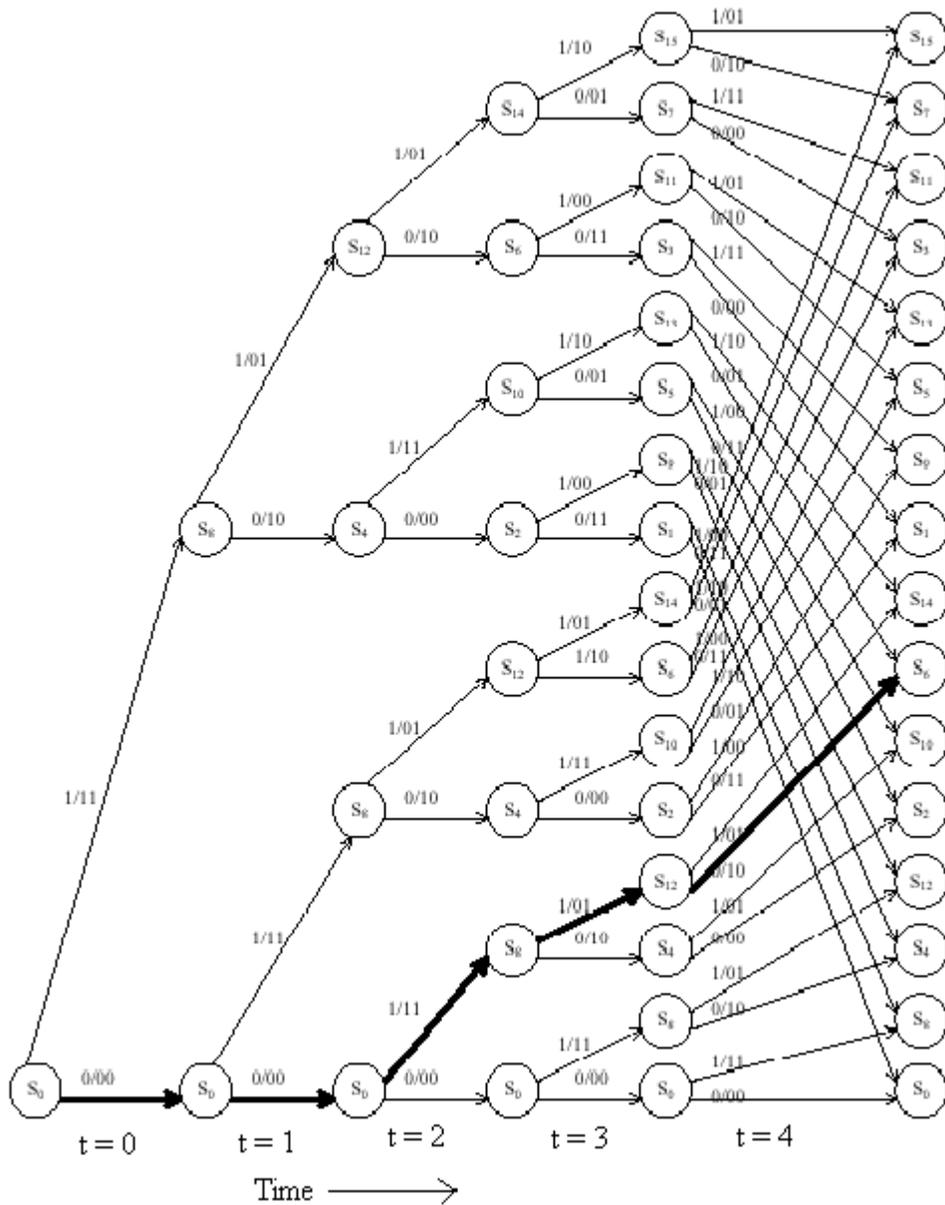


Figure 4.8 Trellis diagram for the GSM Viterbi decoder. Adapted from [6]

At each time interval, the received bits are compared with the inputs to all the states, and the incoming path that has the minimum Hamming distance with the received bits is selected. For example, Figure 4.8 shows the path traced for the received bit sequence 00

00 11 01 10. The transmitted bit sequence along this path is 0 0 1 1 0 which is the decoded sequence.

4.9.2 Speech Decoder

The speech decoder used in the GSM system is the RPE-LTP decoder. Its block diagram is given in Figure 4.9.

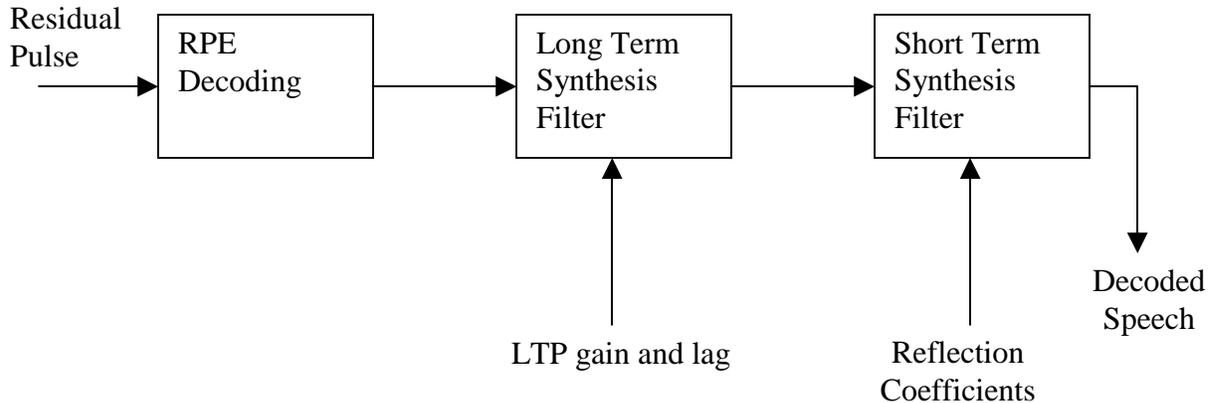


Figure 4.9 Block diagram of the RPE-LTP speech decoder. Figure taken from [26].

The decoder consists of 3 parts – the RPE Decoder, the Long-Term Synthesis Filter and the Short-Term Synthesis Filter. The RPE Decoder decodes the encoded excitation sequence and passes it through the Long-Term Synthesis Filter. This filter then uses the LTP parameters to filter the excitation sequence and the resulting signal is fed to the Short-Term Synthesis Filter, which uses the reflection coefficients to reconstruct the speech waveform.

4.10 Reasons for Selecting GSM System

For this work, we needed a complex system used in the real world. GSM is widely used and the implementation of the encoders and the decoders used in the system is a challenging task. Moreover, the variety of the components in the system enables us to perform hardware-software partitioning to explore alternate implementations. These factors led us to adopt the GSM system as our model.

5 Previous Work

With the background given in the prior chapters, we turn our attention to the work done in the field of GSM modeling, synthesis, SystemC and related areas.

The modeling of a GSM Base Transceiver Station (BTS) is explored at the system level in [35]. One of the four layers of the BTS is the Functional Unit (FU) and this unit covers the set of encoders and decoders that do the data processing. The Unified Modeling Language (UML) is used for modeling the system and the goal of the project is to investigate the usefulness of object-oriented (OO) programming languages to model real life systems.

[36] describes the experience of designing a set top box using an object-oriented methodology based on SystemC. Starting with a high level model of the system written in SystemC, the paper discusses the design guidelines that lead to the system implementation. Another modeling experience with SystemC is detailed in [37]. The authors describe the modeling methodology for implementing the Sobel edge detection algorithm at varying levels of abstraction.

Synthesis from SystemC is described in [38]. A new design environment that supports synthesis from SystemC models is discussed.

In [37], the authors assess the effectiveness of SystemC for system level modeling, and conclude that SystemC has the potential to become a widely used SLML. This work was for a small model. Our work is an extension to this where we consider the modeling and synthesis issues for a large real world system using SystemC. Also, few papers are available that detail the modeling and synthesis with SystemC and CCSC, and hence our work aims to fill that gap.

6 GSM System Modeling

This chapter gives a detailed description of the methodology we used in modeling the GSM system.

The main features of the system that have to be considered while modeling are

- Each module processes one frame of data at a time, and the length of the frame is different for each module.
- Each module has to wait for data from its previous module, process the data and send it to its next neighbor.

Considering these issues, we need to make two decisions – defining the I/O ports for each module, and designing a communication protocol for inter-module data transfer.

6.1 Data Transfer Model

As mentioned before, each unit operates on one data frame, and each frame consists of bits of data. Thus the basic units of processing are individual bits of a frame. These bits can be transferred between modules in different ways.

6.1.1 Bit-Parallel Transfer

At the initial stages of the design, we are not considering any implementation details. Hence we can model the data transfer as transfer of bits from one module to the next, in a parallel fashion. Thus, module N has M output data ports, where M is the number of output bits from this module and the next one, module $N+1$, has M input ports, where M is the number of input bits for this module.

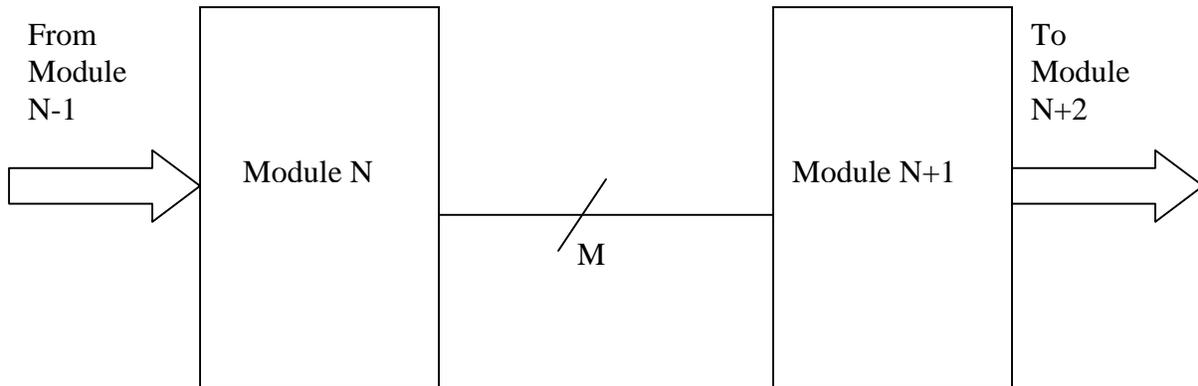


Figure 6.1 Bit-parallel data transfer between two modules

While this model for data transfer is fine for a high-level model, it needs to be refined as we go to the next lower level. This is because, for our system, the value of M is typically high, and it would be cumbersome to have a hardware module with so many ports. For example, the convolutional encoder reads 378 bits and gives 456 bits to the interleaver. Hence, we can use this data transfer model for initial simulation at the highest level of abstraction.

6.1.2 Word Transfer

At the next stage of model refinement, we consider the data transfer issue in more detail. Instead of transferring bits of data in parallel, the data bits can be combined to form words and these words can be transferred between the modules. There is some extra processing at the output of the sender and the input of the receiver to pack the data bits to words and to unpack the words respectively. In addition to packing the bits to words, another refinement is the transfer of these words in a sequential fashion. This ensures that only one data port is needed for each module. The penalty here is the latency, since the transfer occurs sequentially. However, this is not of much importance, since the number of words is small. Considering the example of the convolutional encoder, assuming the size of a word is 16, 24 data reads and 29 data writes are required for each

frame. Moreover, the word size can be set as an adjustable parameter, providing flexibility to the data transfer model.

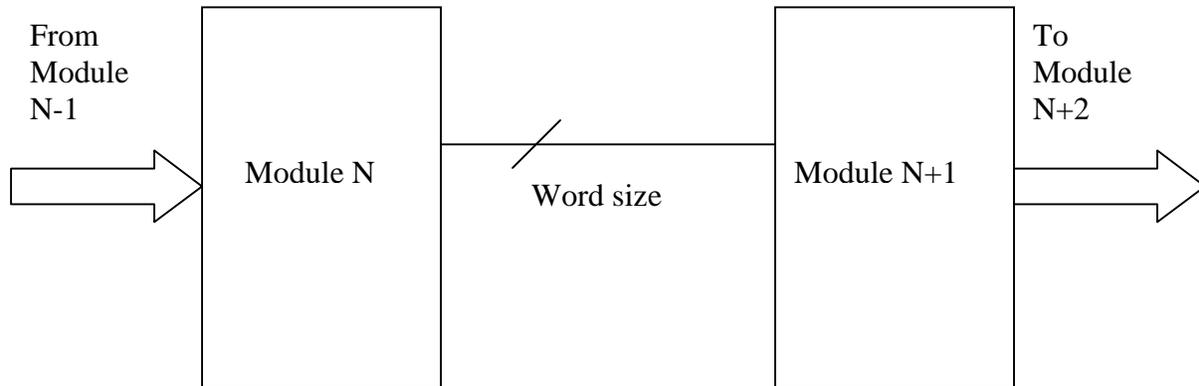


Figure 6.2 Word data transfer between two modules

6.2 Communication Protocol

The problem of inter-module communication is basically a simple one – each module has to wait for data from its previous one, and send it to the next one when that unit is ready to accept data. Examining this issue more closely, we note the following points.

1. When the receiving module is ready to process data, it should indicate that it is ready to receive data, to the sending module.
2. Once it sees that the input data is available, it should read the data.
3. The receiving module has to wait until it gets all the required number of words from the previous one.
4. After receiving the data, the receiver should acknowledge the receipt. In case of sequential word-by-word data transfer, the acknowledgement must be done for each word.
5. When the receiving module has finished processing, it should wait until it receives a signal indicating that the next module is ready to receive data. Now the receiving module is the sending module and the next module is the receiver.
6. If the receiver is ready to receive data, the sender should indicate that the output data is available, and write the data to its output port.

- After writing each unit of data, the sending module must wait for the acknowledgement from the receiving module.

From these observations, the following table can be created, that shows the various control signals required to implement this protocol.

Table 6.1 Summary of the main control signals that implement the communication protocol

Item No.	Signal Name	Direction (Input/Output)	Function
1.	READY_TO_RECV (RTR)	Output	Asserted by the receiving module to indicate that it is ready to receive data
2.	DATAIN_AVAIL (DIA)	Input	Input data is available, asserted by the sender.
3.	RECV_ACK (RAK)	Output	Acknowledgement for data receipt, asserted by the receiver
4.	REQUEST_TO_SEND (RTS)	Input	Receiving module is ready to receive data
5.	DATAOUT_AVAIL (DOA)	Output	Output data is available for the next module. Asserted by the sender.
6.	RECV_ACK_RECVD (RAR)	Input	The acknowledgement from the receiving module indicating that it received data

In addition to these signals, some other signals are also required. These are CLOCK, RESET, START and STOP. CLOCK is the system clock that controls the operation of the system, and RESET is the system reset signal. The system starts operation when START is high, and stops when STOP is asserted. These two signals may be used to control the simulation. The block diagram of a part of the system, showing the inter-

module communication signals is shown in Figure 6.3. The signals CLOCK, RESET, START and STOP are not shown.

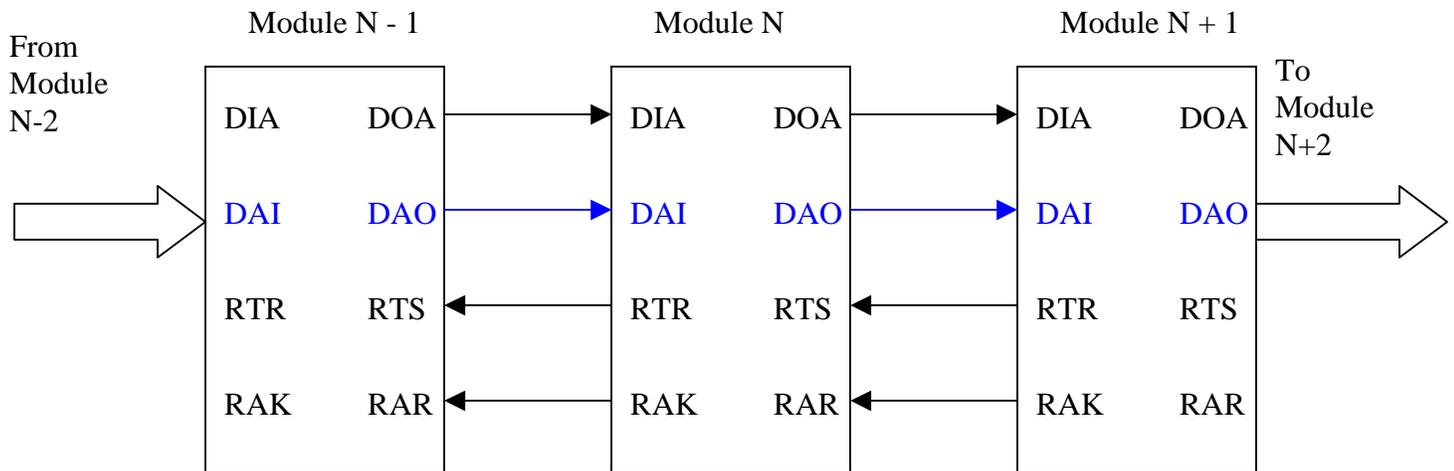


Figure 6.3 Block diagram showing the inter-module communication signals

In Figure 6.3, the signals DAI and DAO stand for DATAIN and DATAOUT, and are colored differently to show that they are not part of the communication protocol. They are the data input and data output pins respectively.

6.3 Module Implementation

In the previous section, the data transfer between the modules was explained. Here we look at the internal implementation details of each module.

Since each module has to accept data, process it and then send the data to the next module, it is logical to start with a simple model with 3 processes – one for input (IP), one for data processing (DP), and one for output (OP). They could execute in a serial fashion, but the real advantage of splitting the model to three parts is that the parts can operate concurrently with the others when required. Thus, to maximize the utilization, it is important to decide the data structures each process operates on, as well as the inter-process communication.

6.3.1 Data Structures

The data frame in each module can be stored in arrays of bits, since arrays are simple and offer support for the required storage and retrieval operations. At the very least, we need two arrays – one to store the input data bits, and one to keep the processed data bits for output. The input array is filled by IP, and the output array is read by OP. DP reads data from the input array and writes to the output array. In the models where word transfer is used between the modules, additional word arrays and processing to convert the word array to bit array and vice-versa are required at the input and output processes.

6.3.2 Inter-process Communication

The input and output arrays are shared between the three processes, and each process should ideally operate independent of one another, if possible. For example, when OP is writing out data to the next unit, IP can read the next frame of data from its previous neighbor. However, IP cannot read the next frame while DP is still operating. We propose a simple protocol based on hand-shaking to resolve these issues. The signals that are used in this protocol are given in Table 6.2.

Table 6.2 Signals used in the inter-process communication model

Signal Name	Driven by process...	Read by process...	Comments
Input_Data_Ready (IDR)	IP	DP	Input data is ready for processing
Input_Data_Processed (IDP)	DP	IP	Input data has been processed
Input_Ack (IACK)	IP	DP	Acknowledgement for the IDP signal.
Input_Ack_Received (IAR)	DP	IP	Acknowledgement for IACK
Output_Data_Ready (ODR)	DP	OP	Output data is ready
Output_Ack (OACK)	OP	DP	OP finished writing the current frame.
Output_Ack_Received (OAR)	DP	OP	Acknowledgement for OACK

The inter-process communication signals are shown in the Process Model Graph (PMG) in Figure 6.4.

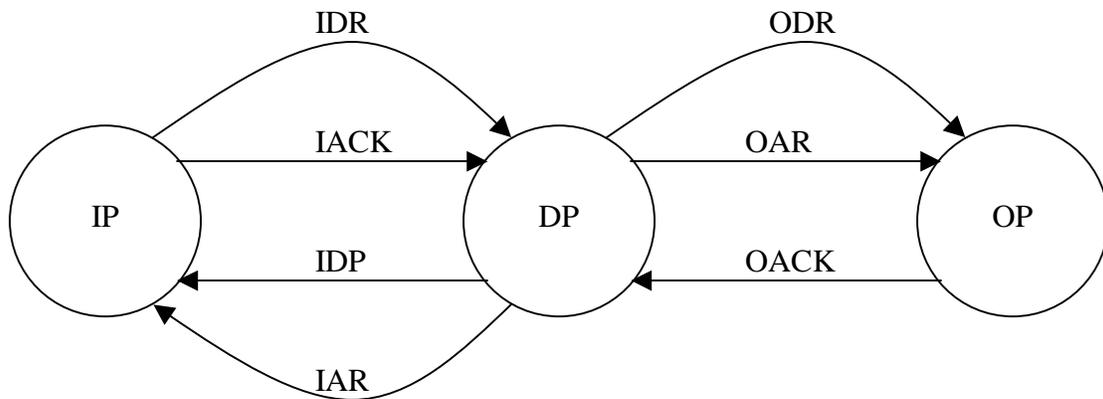


Figure 6.4 PMG showing the inter-process communication signals

The sequence of events that happen in this case is

1. IP reads one frame of data to the input array, and activates IDR.
2. DP checks whether IDR is high.
 - a. If IDR is low, it waits until IDR is high.
 - b. Otherwise, it processes the data, and writes to the output array. Then it sets IDP and ODR to high, IAR and OAR to low.
3. IP waits until IDP is high. Then, it sets IDR to low and IACK to high.
4. DP waits until IACK is high. On sensing that IACK is high, it sets IAR to high.
5. IP waits until IAR is high. When IAR is high, it sets IACK to low. Now IP is assured that the input data frame has been processed, and that it can safely write data to the input array. Hence it proceeds to read the next frame, going to step 1.
6. OP waits until ODR is high. When it is high, OP writes data to the next unit. After writing data, it sets OACK to high.
7. DP waits until OACK is high. Then, it sets ODR to low and OAR to high.
8. OP waits until OAR is high. When OAR is high, it sets OACK to low. Now, OP can proceed dealing with the next output frame, if it is ready. It goes to step 6.
9. DP has now completed processing one frame. It goes to step 2 for the next frame.

This sequence is pictorially represented by the flowchart in Figure 6.5. The lines belonging to each process are colored differently. The signals that implement the communication protocol are shown as dotted arrows connecting the driving block and the sensing block.

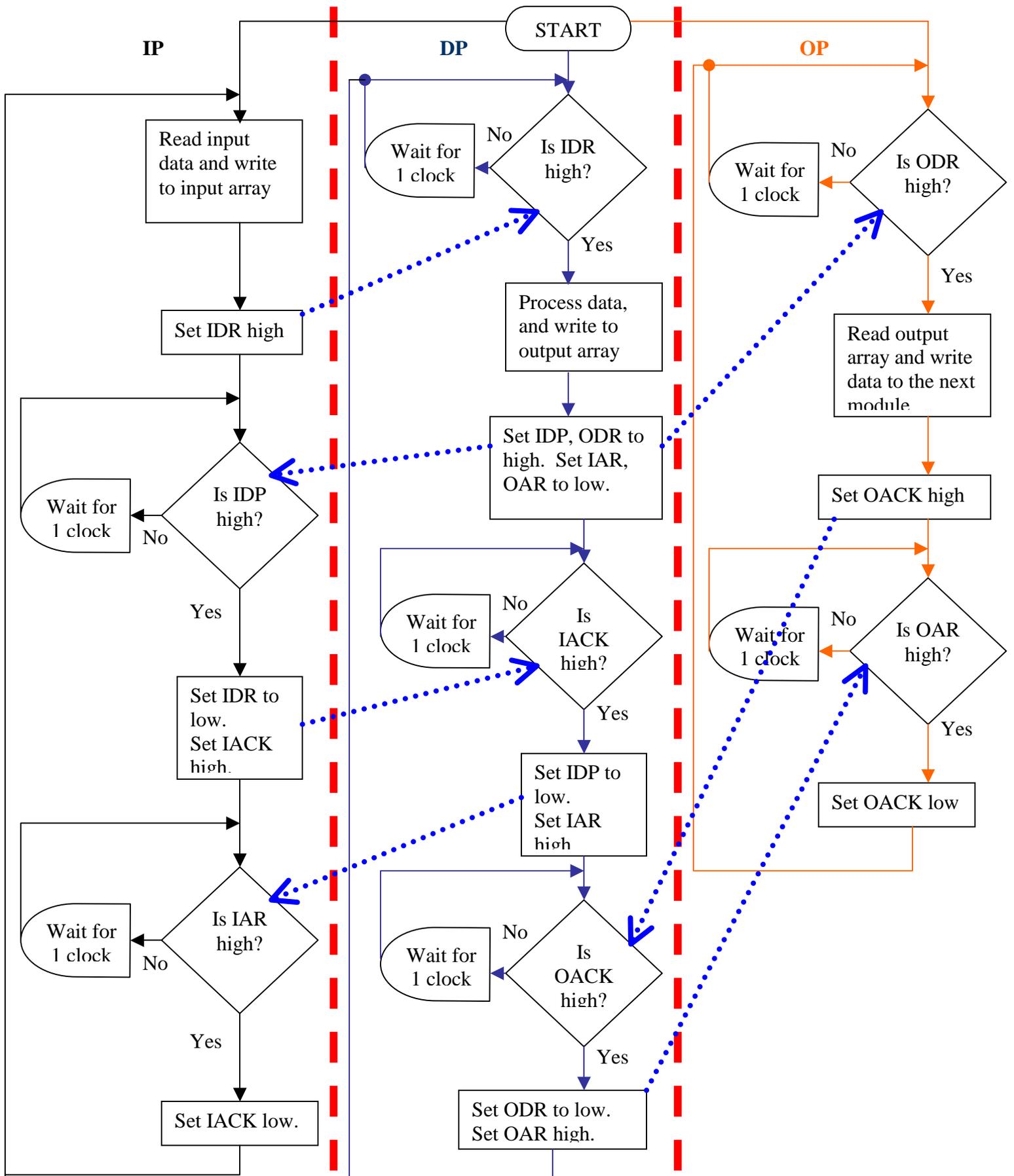


Figure 6.5 Flowchart showing the inter-process communication model

6.3.3 Separation of Communication and Computation

One of the interesting points to note from the above model is that while IP and OP deal with communication with the outside world, DP deals with data processing. DP is not aware of the inter-module communication protocol, and IP and OP do not see the data processing done by DP. This is an example of separating communication and computation. This modeling methodology ensures that the models are scalable. For example, if we want to explore the trade-offs of different communication protocols or data transfer sizes, we modify only the processes IP and OP. Any performance variations can then be attributed to the communication model, since the data processing part (DP) is left untouched. We followed this methodology while refining our communication protocol from the bit parallel data transfer model to the word data transfer model. The refinement process was simple, since IP and OP for each module are similar to the ones in the other modules and are also small in size, unlike DP. Also, if there is a change in the data processing algorithm, modifying only DP ensures that the change is less error-prone compared to a case where the coding style is such that the functionality of all the three processes are written together in one process.

6.4 Process Implementation

We examine the implementation of each process in this section. The three processes are common for each module, and the main difference is in the algorithm for data processing and the input and output array sizes.

6.4.1 Input Process

The function of the input process or IP is to read the data from the previous module and fill the input array. While this is a simple task, we have to note that this process interfaces with the previous module and the DP of the current module. Hence we need to consider the inter-module communication signals that are relevant to this process, as well as the inter-process signals between IP and DP. The pseudo-code for IP can be written as shown in Figure 6.6.

```

Main Loop // for each data frame
  Data count = 0;

  // First, interface to previous unit
  Input Loop // for each data transfer unit (word)
    RAK = false;
    RTR = true;
    Wait until DIA == true;
    Read DATAIN;
    RAK = true;
    RTR = false;
    Wait until DIA == false;
    Increment data count;
    If data count == number of data units in a frame
      Break from Input Loop;
    Else
      Go to Input Loop;
  End Input Loop
  Additional processing, if required, to fill input array.

  // Next, interface to DP
  IDR = true;
  Wait until IDP == true;
  IDR = false;
  IACK = true;
  Wait until IAR = true;
  IACK = false;
  Go to Main Loop
End Main Loop

```

Figure 6.6 Pseudo-code for the input process

6.4.2 Data Process

The data processing part is different for each module. The general flow of this process is represented in Figure 6.7.

```

Main Loop // for each data frame
  // First, data processing
  Read input array
  Process data
  Write to output array

  // Next, interface to IP and OP
  IDP = true;
  ODR = true;
  IAR = false;
  OAR = false;

  Wait until IACK == true;
  IDP = false;
  IAR = true;
  Wait until OACK == true;
  ODR = false;
  OAR = true;
  Go to Main Loop
End Main Loop

```

Figure 6.7 Pseudo-code for the data process

6.4.3 Output Process

The pseudo-code for the output process is shown in Figure 6.8. The output process is similar to the input process, but it operates on the output array. It first communicates with DP to see if the output array is ready, and then writes out the data to the next module. At the next module, it is its input process that interfaces with this output process.

```

Main Loop // for each data frame
  Data count = 0;

  // First, interface to DP
  Wait until ODR == true;

  // Next, interface to next module
  Additional processing, if required, for data transfer
  Output Loop // for each data transfer unit (word)
    Wait until RTS == true;
    DOA = true;
    Write DATAOUT;
    Wait until RAR == true;
    DOA = false;
    Increment data count;
    If data count == number of data units in a frame
      Break from Output Loop;
    Else
      Go to Output Loop;
  End Output Loop

  // Again, interface to DP
  OACK = true;
  Wait until OAR == true;
  OACK = false;
  Go to Main Loop
End Main Loop

```

Figure 6.8 Pseudo-code for the output process

7 GSM Model Synthesis

After developing the GSM models, they were validated using simulation. The next task is the synthesis of the system. This process is iterative with two iterations. We perform an initial synthesis to get the timing, area and power estimates of the system in hardware. This information is used as the hardware metrics by the partitioning algorithm to partition the system to hardware and software. After getting the partitioner results, we do another round of synthesis for the hardware modules, using more accurate delay models to account for the inter-hardware and the hardware-software module communication overheads. In this chapter, we discuss the main issues in synthesis as well as the modeling style for synthesis.

7.1 Synthesis Tool Suite

The synthesis tool suite consists of 3 units

- CCSC – This unit reads the SystemC code and translates the code to netlist.
- Design Compiler (DC) – This is the user interface of the synthesis tool. The scheduling operations are performed by this unit.
- BCView – After scheduling, the design can be evaluated for refinement using this GUI tool. A detailed description of this unit is given later in this chapter.

7.2 The Synthesizable Subset of SystemC

SystemC can be used to model abstract concepts, and not all such models have equivalent hardware representation. Sometimes a construct does have a valid hardware representation, but it may be very complicated. The idea of virtual functions is an example of the former while the concept of file I/O provides an illustration of the latter. Because of these reasons, the synthesis tool supports only a subset of valid SystemC constructs. We list only those constructs that affect our model; a complete list of synthesizable constructs can be found in [39].

7.2.1 Data Types

The data types that are supported and which are relevant to our models are

- Integer data types – short, int, long, and the SystemC integer types, including sc_int and sc_uint
- Boolean data type, bool
- Arrays

Since the GSM system operates on arrays of individual bits and integers that are stored in arrays, this subset is adequate for our model.

7.2.2 Language Constructs

Of the three constructs in SystemC that are used to model hardware-related processes, only the clocked thread process is supported for synthesis. This means that the models operate in a synchronous fashion, with control signals sampled at the clock edges. Conditional execution, loops, and function calls are also supported by the synthesis tool.

7.3 Memory Mapping

One of the main issues in synthesizing a large design is the way arrays are implemented by the synthesis tool. Arrays are basically storage units, and the default method is to map the arrays to registers. Dedicated hardware is allocated for every read and write operation to these registers. Hence, when the array size is large, the overhead caused by the logic for array read and write can be quite high. This results in long time for the synthesis tool to finish execution.

To overcome this problem, the synthesis tool has a feature where the user can choose one of two possible array implementation techniques. The first one is mapping arrays to register files. Register files are collections of registers where the read and write operators are built on the fly by the synthesis tool. The second option is to map arrays to memories. A memory has built-in read/write logic, hence dedicated hardware to perform these operations are not created by the synthesis tool. This option is more efficient than mapping arrays to register files. A comparison of array implementations is summarized below.

Table 7.1 Comparison of array implementations

Parameter	Register File	Memory
Read / Write logic	Requires the generation of array read and write operators	Read and write operators are built-in.
Area	Higher	Lower
Computational complexity of the tool	Higher	Lower

From the above table, it is clear that the mapping arrays to memory is the best option. However, to do this, the user must have the model of an appropriate memory module, and also its timing information. Moreover, an interface has to be created for the synthesis tool to use the memory.

To ease the process of creating the memory interface, a GUI tool named *memwrap* is available. In addition to creating memory interface for vendor memory modules, it also allows the creation of ‘dummy’ or exploratory memory graphically and generates all the associated files automatically. The term ‘memory wrapper’ is used to denote the memory interface. One of the important features of this memory wrapper tool is that several parameters of the memory can be easily controlled by the user.

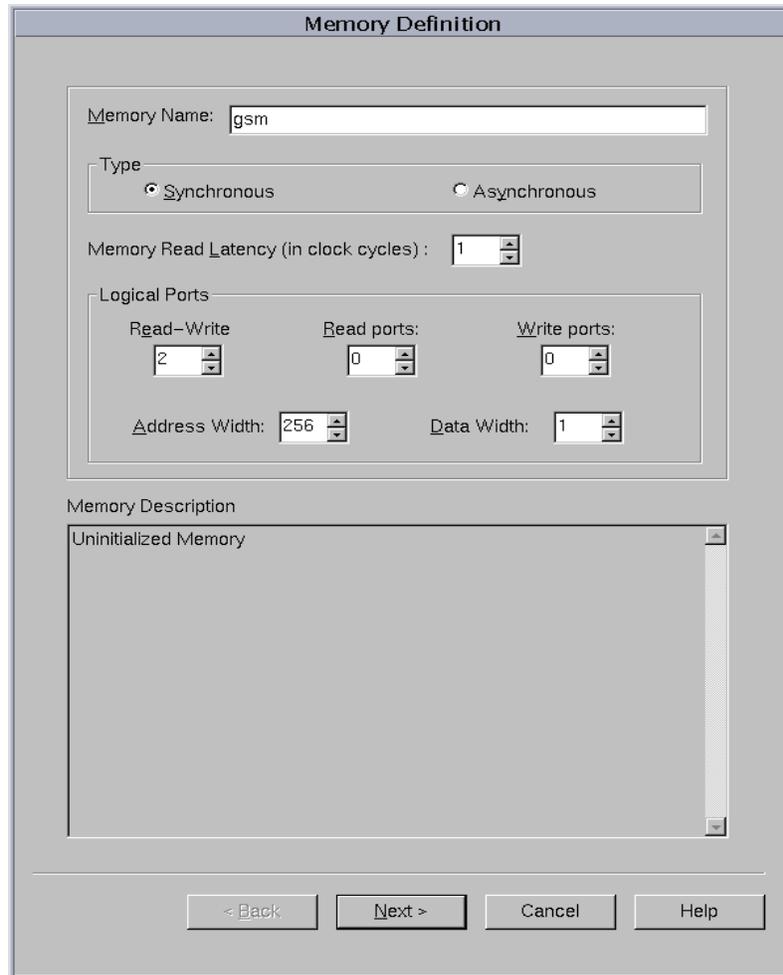


Figure 7.1 Main window for creating exploratory memory in memwrap

Figure 7.1 shows the primary window of memwrap for exploratory memory wrapper creation. It can be seen that the user has flexibility in controlling parameters like number of read/write ports, read latency, address and data widths, etc. Each of these parameters plays an important role in the final synthesized design. For example, if the memory read latency is set to a value 2, and if an array is mapped to this memory, all reads from this array takes 2 clock cycles each. This directly affects the performance of the system. Also, the address and data widths must be decided considering the length of the array and the data type of the value stored in it. Thus, generating memory wrappers is an important consideration in synthesis.

A useful feature of the synthesis tool is that multiple arrays can be mapped to the same memory, if the memory size is at least as high as the sum of the array sizes. This means

that we can create a large memory module and map arrays of the same type to the same memory. In our models we use large arrays, often of the same data type, extensively in the speech encoder, speech decoder and the Viterbi decoder modules, and hence we used memwrap to create appropriate memory wrappers. The parameters for the wrappers used are given below.

Table 7.2 Memory parameters used for synthesis

Memory	Ports	Read latency	Address size	Data size
Speech encoder	1 (read/write)	1	2000	16
Speech decoder	1 (read/write)	1	1000	16
Viterbi decoder	1 (read/write)	1	20000	1

7.4 Clock Frequency Selection

In all synchronous systems controlled by a clock signal, the clock frequency has to be specified. The GSM specification does not mention anything specific about clocks, since these details are implementation dependent. Hence we had to take into account other issues while deciding the clock frequency. Since we target our design to an architecture consisting of the StarCore DSP as a component, it was decided that the hardware modules would also be using the same clock as this processor. The clock frequency was thus fixed at 300 MHz. This decision was done to simplify the overall system, since managing multiple clocks can be a non-trivial task.

Though this decision was a simple one, it was found that it affects the synthesis process. The library components used by the tool for mapping the model to hardware have fixed delays. If the scheduler cannot schedule an operation in one clock cycle, it makes that operation a multi-cycle one, spanning several clocks. The synthesis tool uses special rules to schedule these operations and their interaction with other operators, and the scheduler reports errors if the scenario cannot be realized. The easiest solution to overcome these errors is to increase the clock period, to convert the multi-cycle operators to single-cycle ones that are simpler to schedule. When the clock frequency is already determined, this solution is not an option, and re-writing the code is required to alter the interactions of the multi-cycle operators with the other operators.

We experienced scheduling errors like these, and had to rewrite the code to get around the problem.

7.5 Scheduling Modes

The synthesis tool allows the user to choose one of two possible scheduling modes – cycle fixed mode and super-state fixed mode. The concept of using wait () statements is an important idea behind these scheduling modes.

7.5.1 Importance of wait () statements

In SystemC, wait () statements in the code cause the simulator to wait until the next clock cycle. Hence if two operations are separated by a wait () statement, they occur on different clock cycles, and the wait () statement can be considered equivalent to a clock edge. Another way to look at this is to consider the code as a Finite State Machine (FSM). The code between two wait () statements describes one state and the clock edges or wait ()'s cause state transitions. This idea is shown in Figure 7.2.

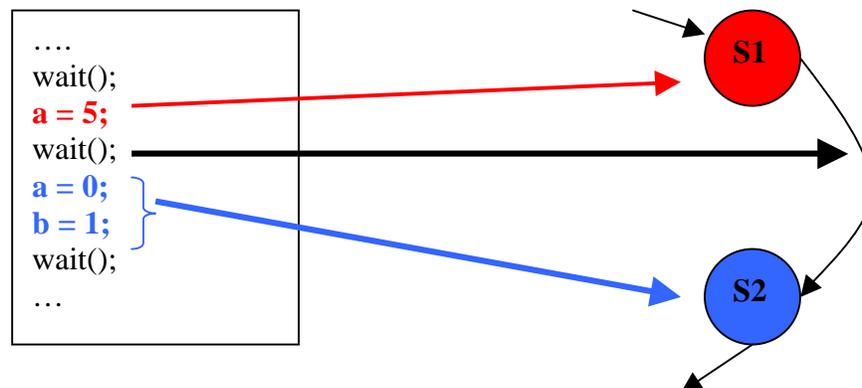


Figure 7.2 SystemC code (left) and the corresponding FSM representation (right).

The two scheduling modes differ in the way the wait () statements are mapped to state transitions.

7.5.2 Cycle Fixed Mode

In this mode, the mapping between the wait ()'s and the state transitions is one-to-one. In other words, the code between two wait ()'s constitute an atomic state, as represented in the above diagram. This is shown in the example in Figure 7.3.

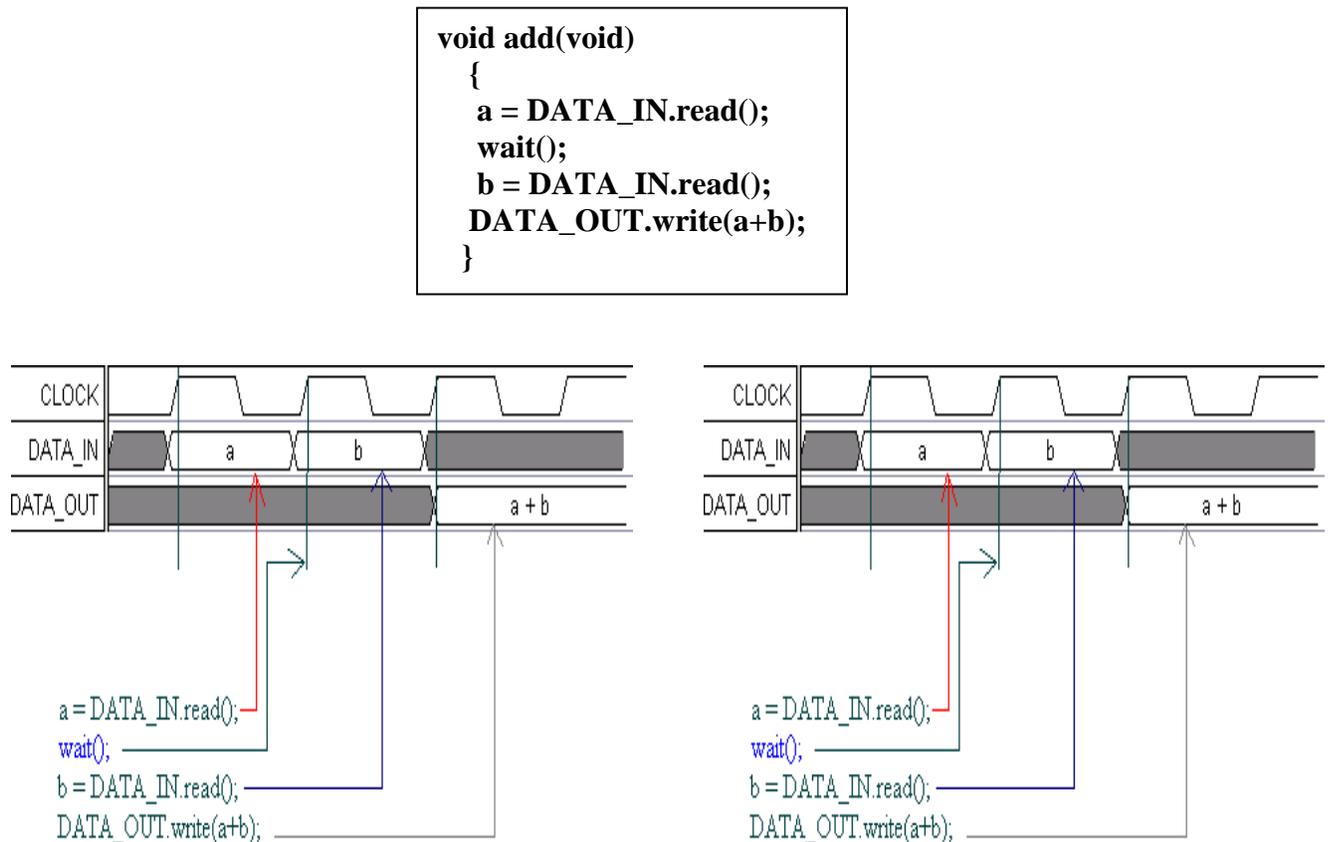


Figure 7.3 Cycle fixed scheduling mode. SystemC code (above) with the corresponding pre-synthesis simulation result (left) and the post-synthesis simulation result (right)

It must be noted that though there are no wait ()'s separating the last two statements – read from `DATA_IN` port and write to `DATA_OUT` port – there is a delay of one clock cycle between them. This is because all writes to ports are registered in SystemC and hence are delayed by one clock cycle.

Here the pre- and post synthesis simulations match exactly, and the user has accurate control over the scheduling process. While this is an advantage, it puts some restrictions on the part of the user. These are

- The user must decide the FSM in advance, and insert wait () statements accordingly.
- The execution delay of the hardware components in an atomic state must be less than one clock period. Otherwise, the state machine is not realizable, and the tool reports a scheduling error.

It is important to note that in practical cases, the second point discussed above is difficult to satisfy. The synthesis tool uses a library of hardware components for mapping code to hardware, and the user must have knowledge of the timing details of all the components in an atomic state. The clock period is usually decided independent of these timing details, and it may not be possible to ensure that the clock period is sufficiently high to complete the execution of one state before changing to the next. Because of these difficulties, the super-state fixed scheduling mode is provided.

7.5.3 Super-state Fixed Mode

Here, the mapping from wait () statements to state transitions is one-to-many. The code between two wait ()'s constitutes a "super-state" that consists of one or more atomic states. Hence the synthesis tool inserts new states and state transitions, if necessary, in addition to the ones implied by the wait () statements. While this ensures that the clock period and the hardware delays are not tightly related, it means that the pre- and post synthesis timing may not match, as illustrated by the example in Figure 7.4.

```

void add(void)
{
  a = DATA_IN.read();
  wait();
  b = DATA_IN.read();
  DATA_OUT.write(a+b);
}

```

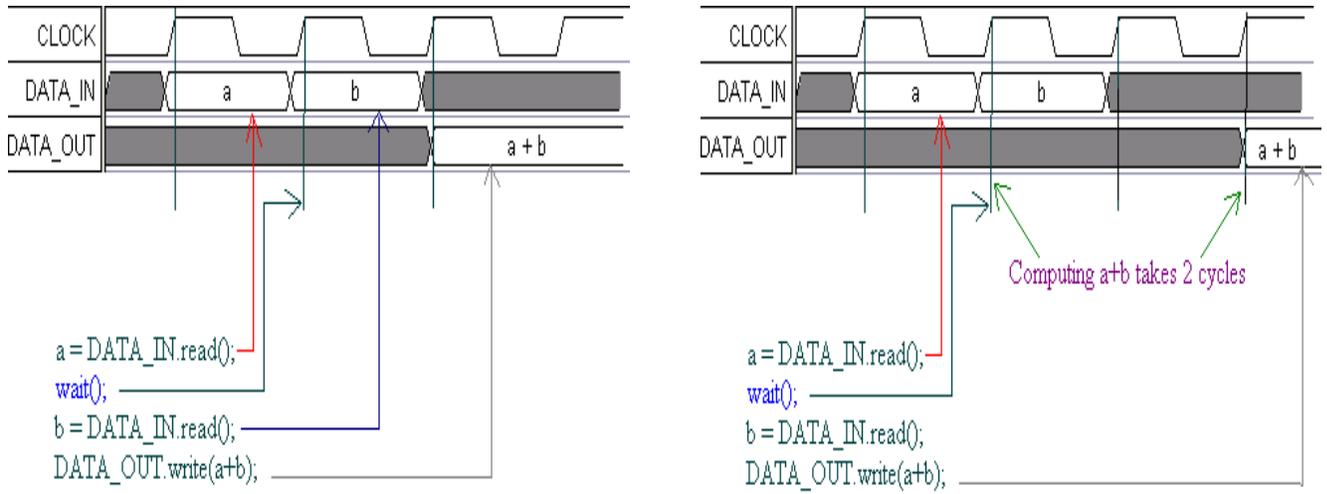


Figure 7.4 Super-state fixed scheduling mode. SystemC code (above) with the corresponding pre-synthesis simulation result (left) and the post-synthesis simulation result (right)

Also, the user can write the code without using wait () statements, so that the synthesis tool has more options to explore the design space. The consequence of this is that the tool may take a long time to execute, since it has to evaluate more possibilities.

The above considerations led us to use the super-state fixed scheduling mode for our system.

7.6 BCView

BCView is a GUI tool that allows the user to analyze the scheduled design. This tool gives detailed information about the design, and the information is organized in different windows. The windows are HDL Browser, FSM Viewer, Reservation Table, Selection Inspector and Scheduling Error Analyzer. All the windows are cross-linked so that selecting an item in one window highlights the corresponding entries in the others.

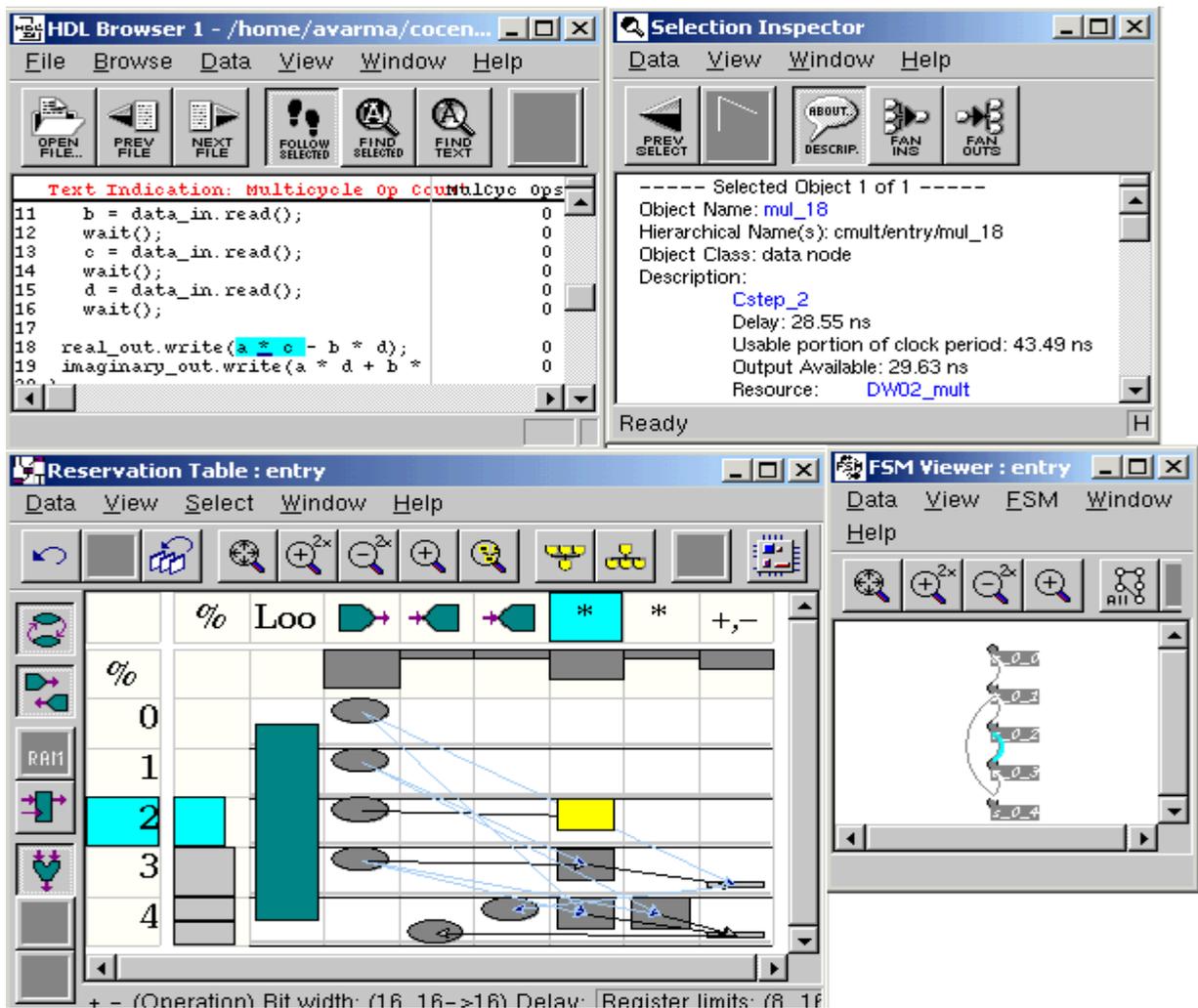


Figure 7.5 BCView showing the different windows

7.6.1 HDL Browser

This window shows the SystemC code for the design. An example is shown in Figure 7.6.

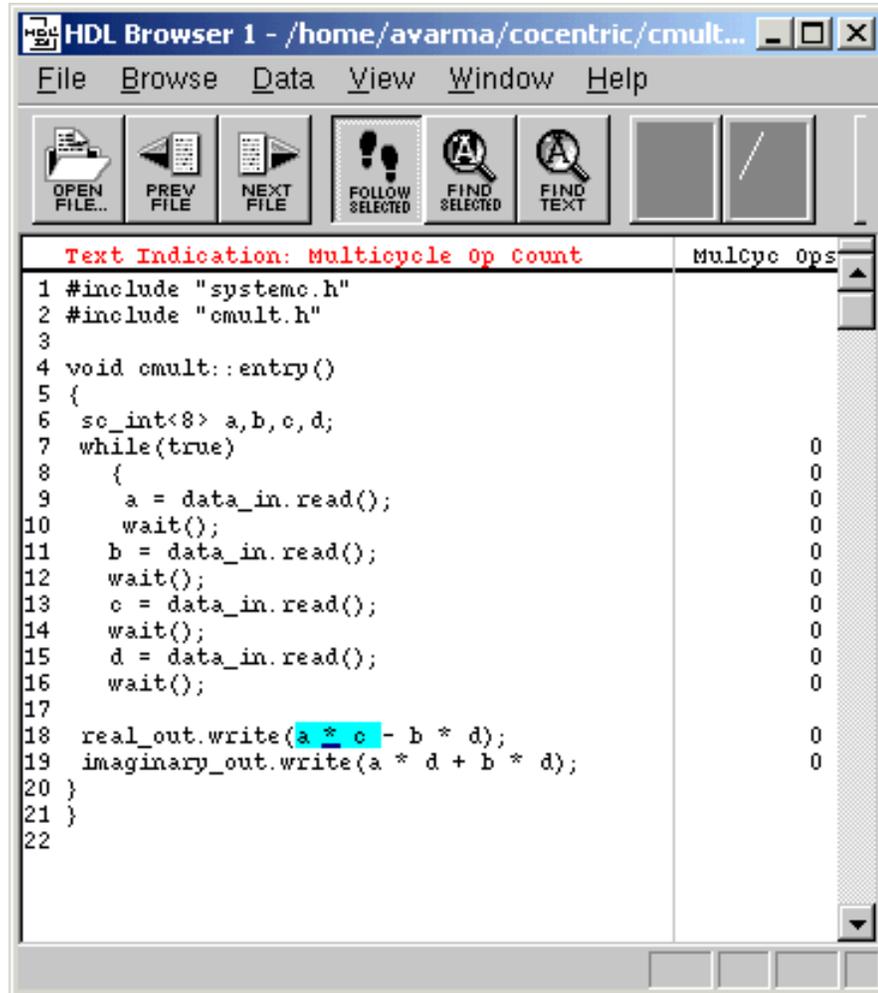


Figure 7.6 HDL Browser window

7.6.2 FSM Viewer

The FSM Viewer window represents the FSM model of the system. The user can follow the state transitions in this window and verify that the SystemC code is executed in the expected order, by looking at the HDL Browser window.

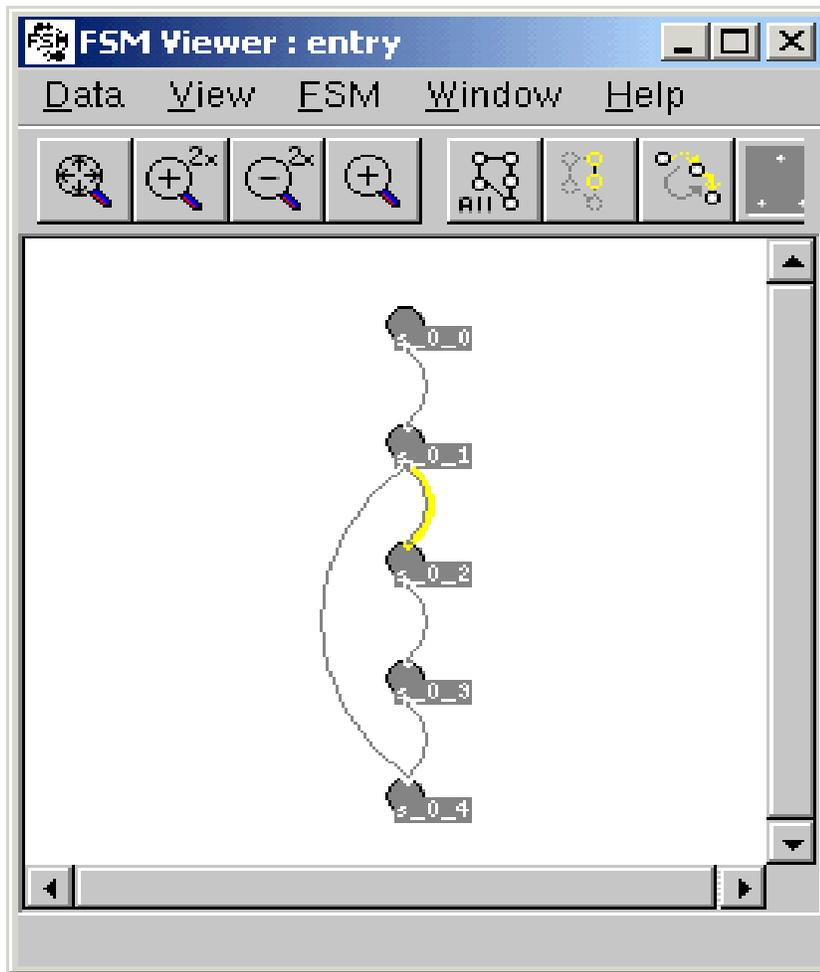


Figure 7.7 FSM Viewer window

7.6.3 Reservation Table

The scheduling details of the design are shown in this window. In addition to the resource allocation for each clock cycle, parameters like clock utilization, resource latency, data dependency, etc. are also indicated.

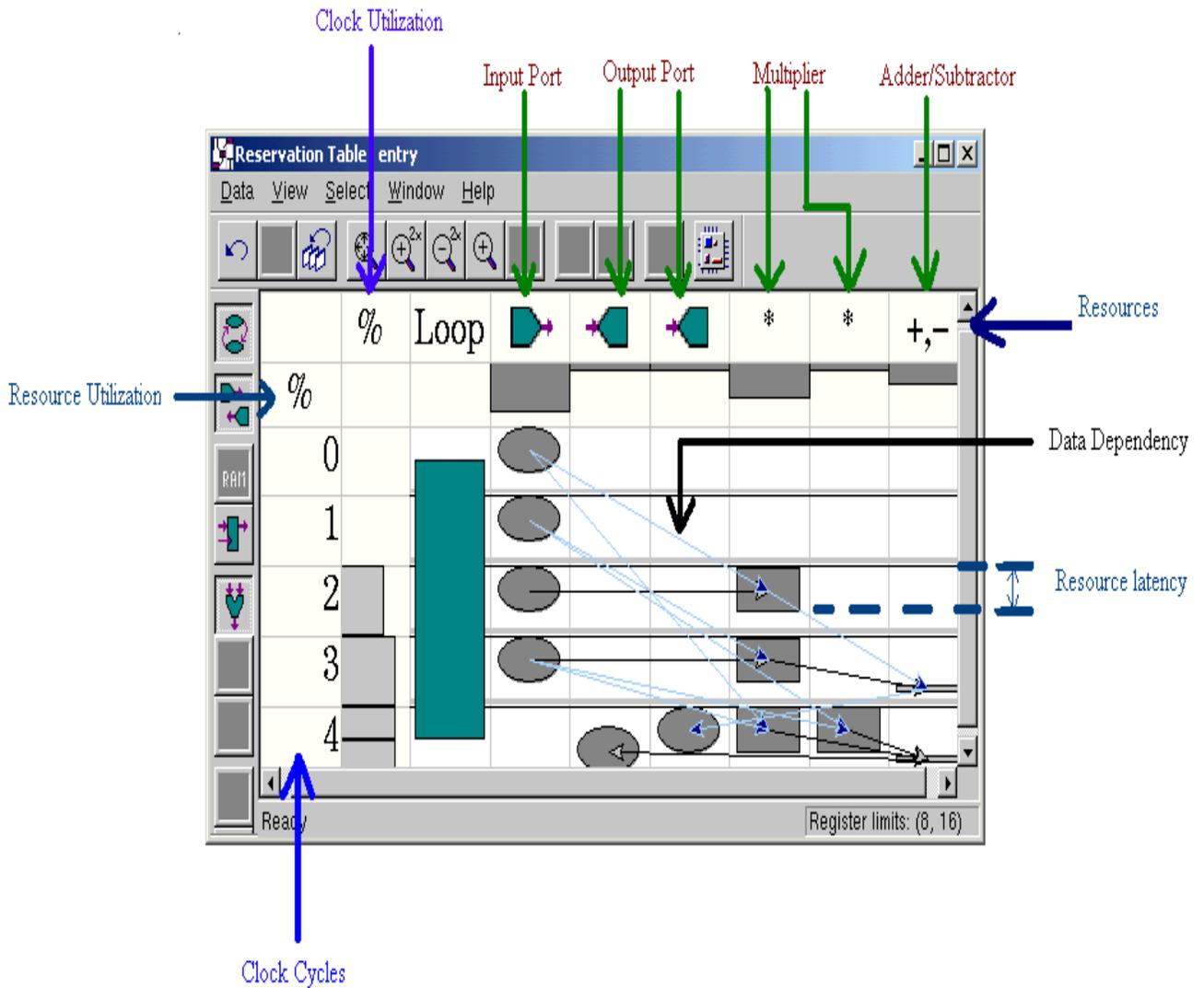


Figure 7.8 Reservation Table window

7.6.4 Selection Inspector

Details of entities selected in other windows are described here. Figure 7.9 shows an mple.

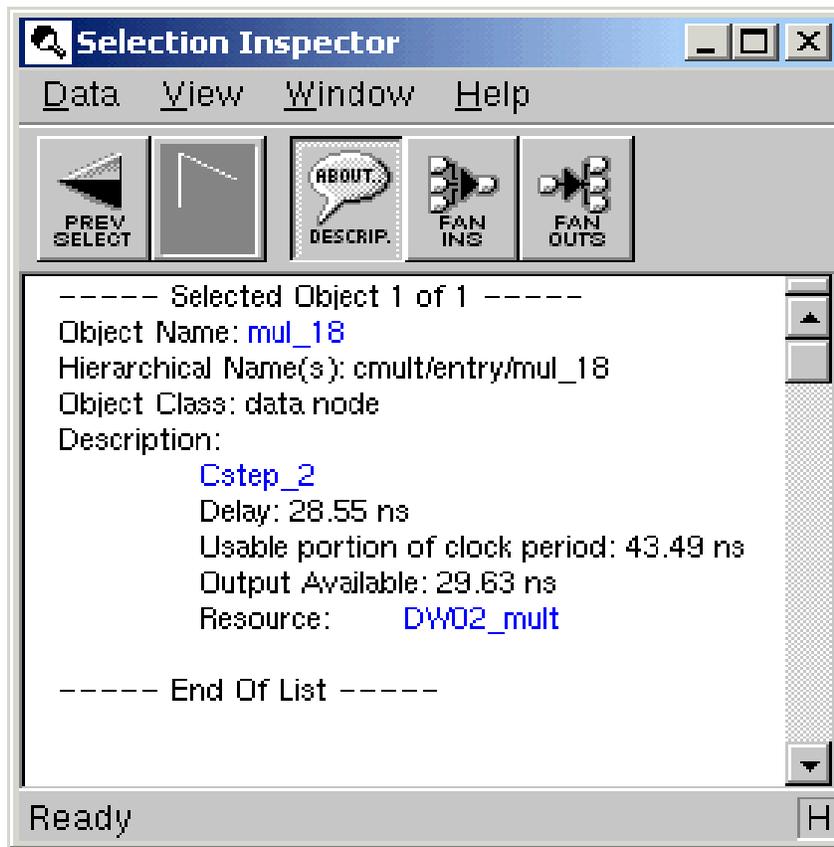


Figure 7.9 Selection Inspector window

7.6.5 Scheduling Error Analyzer

When there are scheduling errors, BCView displays this window, along with HDL Browser and Selection Inspector. The Selection Inspector gives information about the errors and suggests possible corrective action.

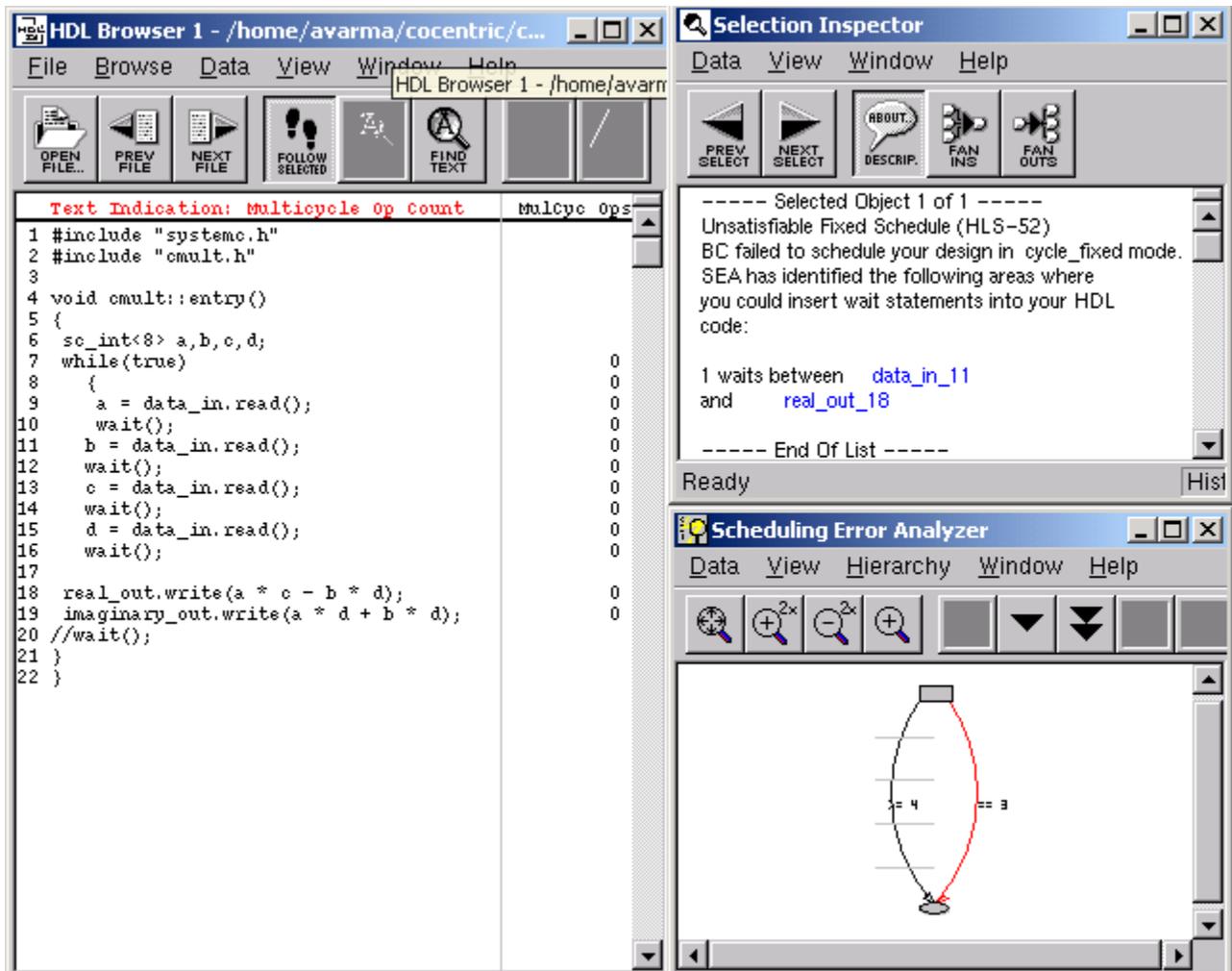


Figure 7.10 BCView with Scheduling Error Analyzer

7.7 Memory and Tool Issues

The modules speech encoder, speech decoder, and Viterbi decoder perform extensive computation and since we used the super-state fixed scheduling mode, the tool had to explore many alternative designs. This resulted in system memory overflow, and hence we decided to split each of these modules to smaller sub-modules. The speech encoder was divided to 5 parts, the speech decoder, to 3 parts, and the Viterbi decoder to 2 sub-modules. It was then verified that the system functionality remains the same.

Even after partitioning these modules, it was found that we could not synthesize all the sub-modules. This was because of some issues with the tool and these were reported to

the vendor. One of the issues was due to the fact that we were using an older version of the tool, and our problem was addressed in the latest version. But this version was released when we were in the final phase of our project and hence could not use it for our system. Another issue was submitted as a bug report.

8 Results

The results of the work done for this project are presented in this chapter. Since the work involved modeling the GSM system and synthesis, we provide the results for these sub-tasks under separate categories.

8.1 GSM Modeling

To simulate the GSM system, we used sound samples, and passed these through our system. The sound samples were recorded and saved as files in 'au' format. The reconstructed sound at the receiver was also saved to file, and compared with the original sample.

We performed the experiment with different types of sound samples. The samples we considered are listed in Table 8.1.

Table 8.1 Sound samples used for testing the system

Sound sample name	Description
Male Speech	Male speech sample
Female Speech	Female speech sample
Music	A sample of instrumental music
Tone	A sine wave (tone) of frequency 1 kHz

We first transmitted a sound sample through the system without introducing any error at the channel. The sound sample was a male speech sample. The time domain waveform for this experiment is shown in Figure 8.1.

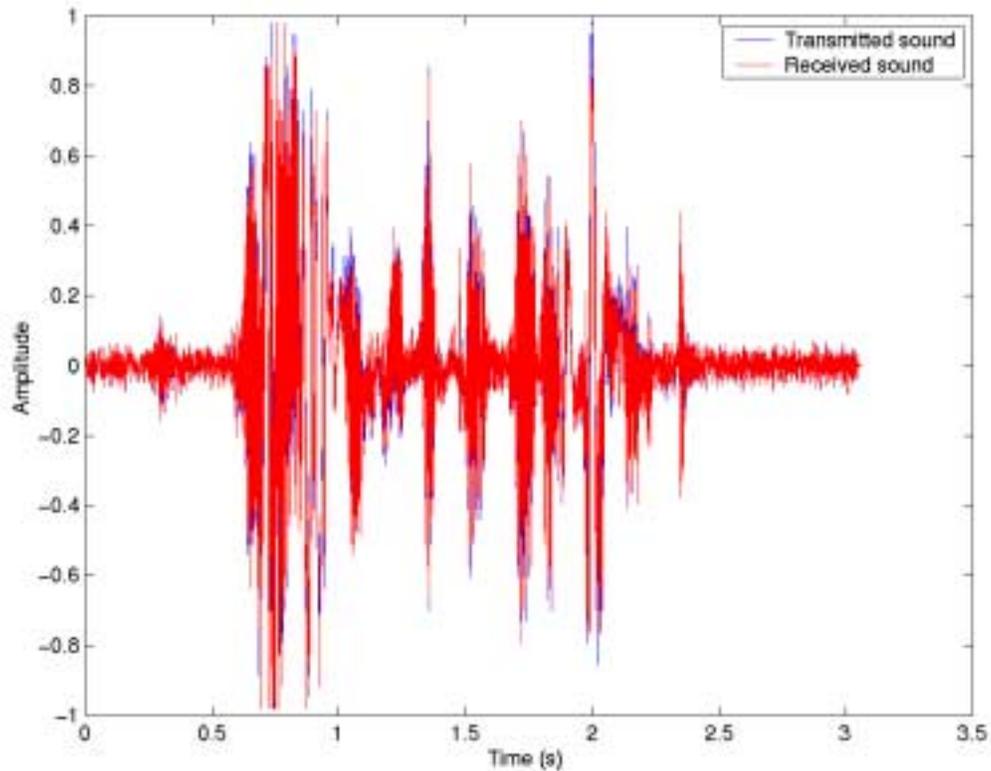


Figure 8.1 Transmitted and received sound samples for Male Speech

The time domain waveforms indicate that even with no errors introduced by the channel, the received waveform differs from the transmitted one. The Mean Squared Error (MSE) between the two samples was found to be 0.0014. This is because the speech compression is lossy, but since we transmit speech and not data, this is not important, as the difference is barely audible.

The frequency domain waveforms of the transmitted and the received samples are presented in Figure 8.2.

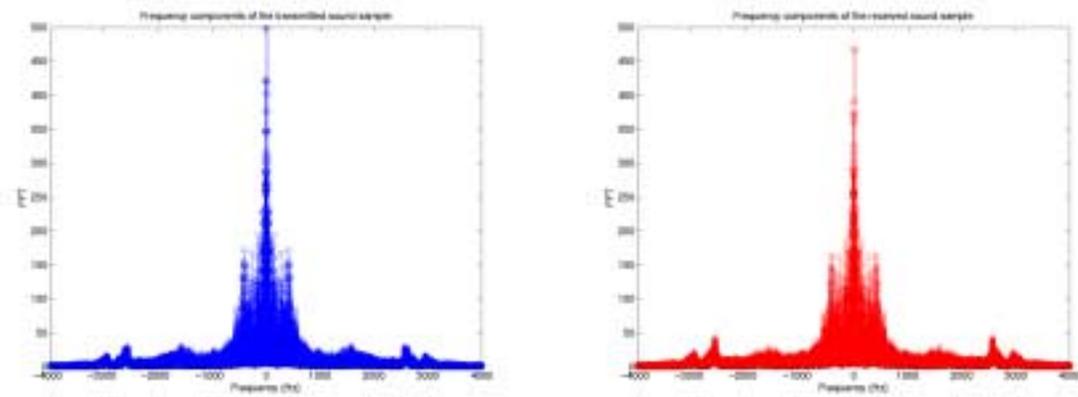


Figure 8.2 Frequency components in the transmitted sound (left) and the received sound (right) for **Male Speech**

The results for the other sound samples are presented below.

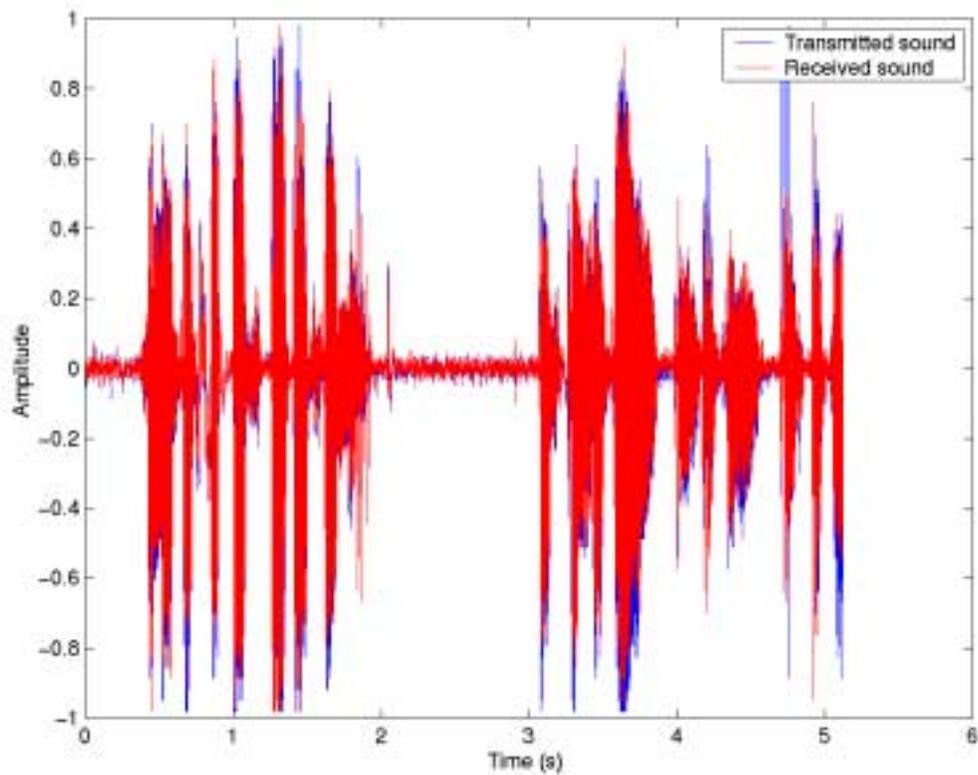


Figure 8.3 Transmitted and received sound samples for **Female Speech**

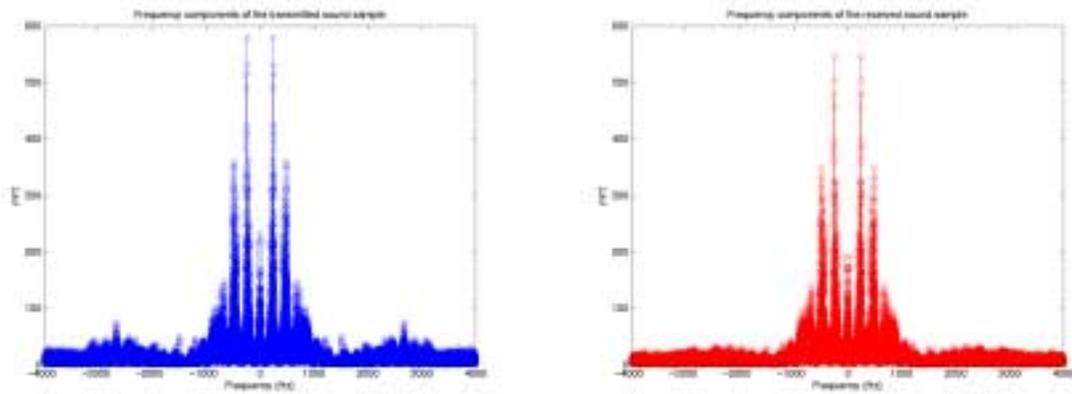


Figure 8.4 Frequency components in the transmitted sound (left) and the received sound (right) for Female Speech

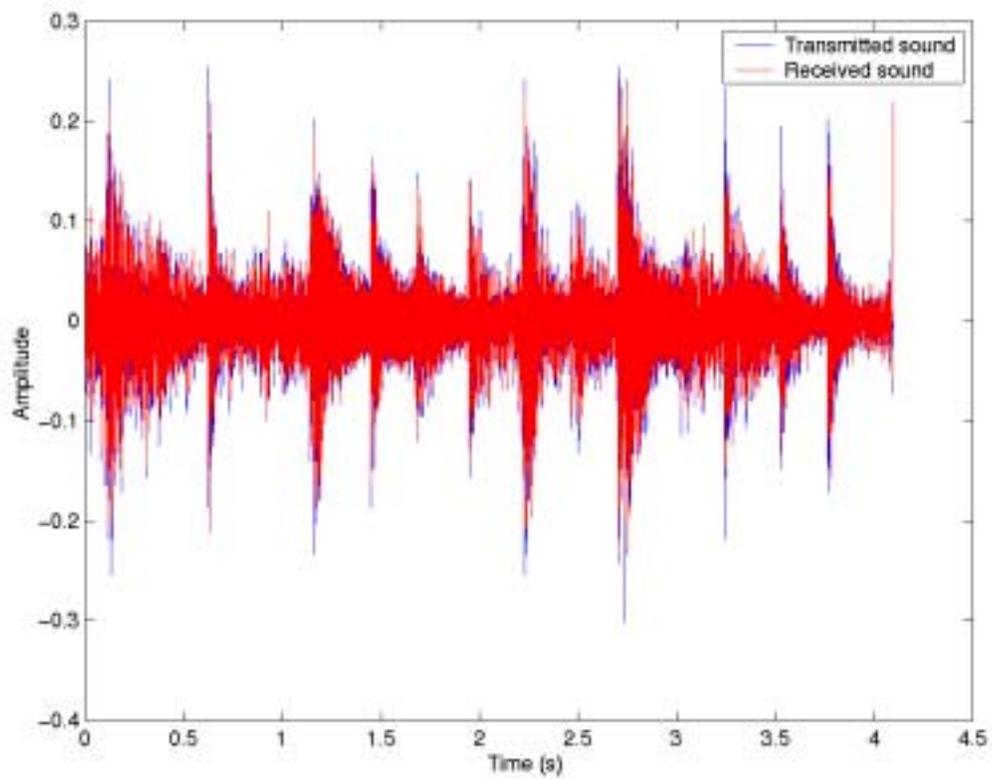


Figure 8.5 Transmitted and received sound samples for Music

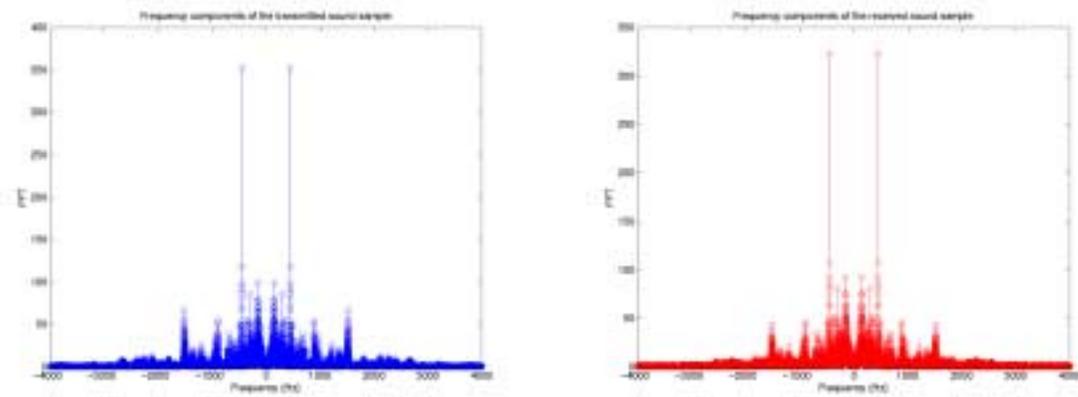


Figure 8.6 Frequency components in the transmitted sound (left) and the received sound (right) for Music

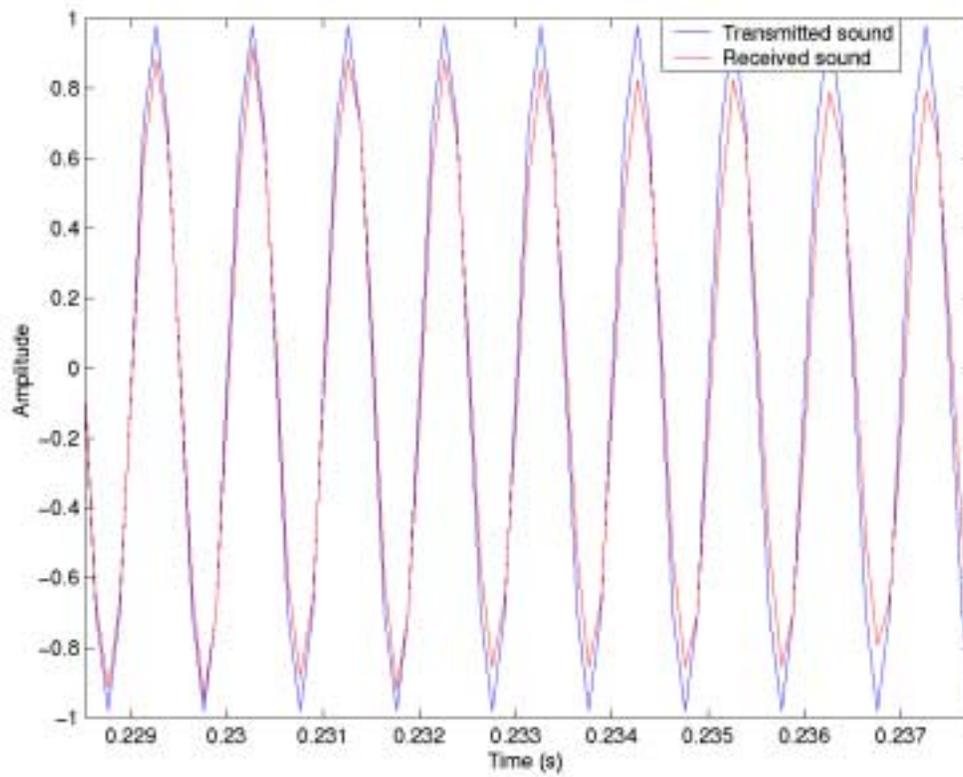


Figure 8.7 Transmitted and received sound samples for Tone

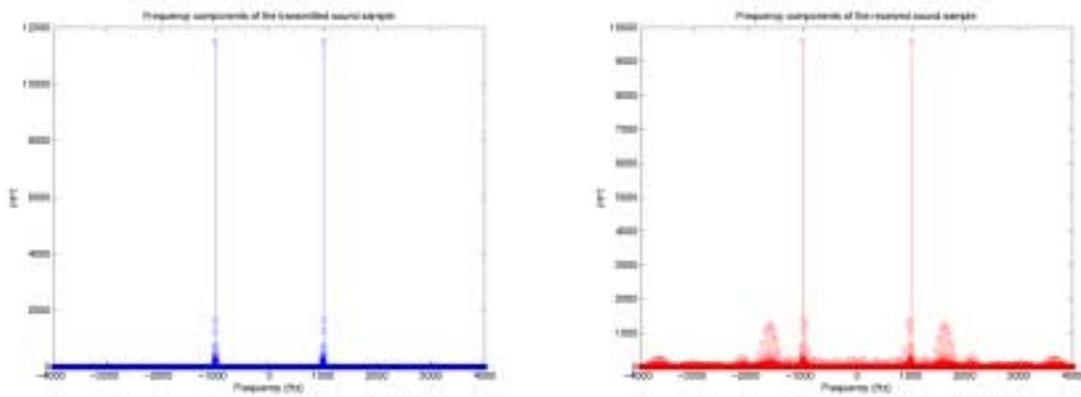


Figure 8.8 Frequency components in the transmitted sound (left) and the received sound (right) for **Tone**

The MSE for the sound samples are given in Table 8.2

Table 8.2 MSE for sound samples

Sound sample name	MSE
Male Speech	0.0014
Female Speech	0.032
Music	0.0016
Tone	0.4846

It can be observed that the MSE for the tone is much higher than that of the other samples, indicating that the GSM sound compression is more suited for speech data.

The same experiment was performed with different values for the channel bit error probability for the male speech sample. Some of the results are summarized in Table 8.3.

Table 8.3 Summary of simulation results

Bit Error Probability (in parts per 10000)	Burst length (bits)	Total Bits	Actual Error (bits)	MSE
0	1	91168	0	0.0014
1	1	91168	7	0.0014
5	2	91168	92	0.0137
10	5	91168	440	0.0221
50	10	91168	4210	0.1661
100	10	91168	8250	0.1668

From the table, it may be observed that as the bit error probability increases, the MSE also increases. This indicates that the error correction capability of the system breaks down with large number of errors. In practical systems, the error correction capability can be made higher by using soft-decision Viterbi decoder, which uses the analog values from the demodulator for bit estimation.

8.2 Synthesis Results

The synthesis results give the timing, area and power estimates for the hardware part. These are summarized in Table 8.4. All the area values in the table are in terms of 2-input NAND gate counts.

Table 8.4 Synthesis results showing the timing, area and power for the hardware modules

Module	Clock Cycles (Period = 3.33 ns)	Combinational Area	Sequential Area	Total Area	Power (μW)
A5 Decoder	3555	2168	3861	6029	74.9187
A5 Encoder	3555	1625	3410	5035	74.9187
Convolutional Encoder	3162	2158	3630	5788	236.5699
Differential Decoder	1179	168	396	564	44.1643
Differential Encoder	1326	832	418	1250	49.4633
Interleaver Decoder	22706	8486	1617	10103	167.6485
Interleaver Encoder	21035	6012	24662	30674	828.6318
Packet Decoder	1599	1081	484	1565	60.2614
Packet Encoder	1301	7835	1617	9452	173.6756
Parity Decoder	4673	4770	4246	9016	177.5278
Parity Encoder	2625	4855	3971	8826	162.9159

The values for the speech encoder, decoder and the Viterbi decoder could not be obtained due to the issues with the tool discussed in chapter 7.

The results show that though there seems to be a similarity between the timing, area and power details, it is not always the case. For example, the packet encoder has a slightly lower latency but its area is much higher compared to the parameters for the packet decoder. If we partition our system to hardware and software components based on timing information, we may find one partition where the packet encoder and the packet decoder are either both hardware or software units, since their latency values are very close. However, if the partitioning algorithm is modified so that hardware-software partitioning is based on area, we may find that these units are in separate partitions, because of the vast difference in their areas. Hence partitions based on area and power may be different from the one based on timing information.

9 Conclusions and Future Work

In this thesis, we discussed the experience of modeling a GSM system using SystemC and synthesizing it using CCSC. We started our modeling task at a high level of abstraction and refined the model to perform word-based data transfer. The communication protocol was also developed. Though the synthesis tool was in the initial stages of development, we could synthesize most of our design.

SystemC proved to be a good language for our modeling task. We could make executable specifications with low simulation times, and also traverse the design hierarchy by refining our model in an iterative manner. Since the language is based on C++, learning the language was not a problem. The support for SystemC in the EDA industry is also widening, indicating the possibility of more SystemC-centric tools in the near future. The support for SystemC synthesis is growing with the introduction of new versions of CCSC and other tools, and it could become popular in a few years. We believe SystemC is a good SLML.

Currently work is under way to set up an Internet site where users can use our model and run it with their speech samples. In the future, the model we developed could be synthesized completely with the latest versions of SystemC and CCSC. The next step would be to fabricate the entire system in a single chip, and test it in the real world.

We tested the usefulness of SystemC for modeling a system in the communications domain. Work may be done to model systems from the other application areas like image processing, computer networking, etc. to evaluate the performance of the language and identify any scope for improvement.

References

1. Semiconductor Insights Inc, “A Structural Analysis of the Intel 1.5 GHz Pentium 4 Microprocessor”,
http://www.semiconductor.com/reports/search_detail.asp?device=5071&report=1530 (current November 2001)
2. Intel, “Intel Microprocessor Hall of Fame”,
http://www.intel.com/intel/intelis/museum/exhibit/hist_micro/hof/hof_main.htm
(current November 2001)
3. <http://ls12-www.cs.uni-dortmund.de/~niemann/codesign/codesign.html> (current November 2001)
4. Open SystemC Initiative, “SystemC Homepage”, <http://www.systemc.org> (current November 2001)
5. Synopsys Inc, “Synopsys CoCentric SystemC Compiler”,
http://www.synopsys.com/products/cocentric_systemC/cocentric_systemC.html
(current November 2001)
6. X. Qu, *Automated Testbench Generation for Communication Systems*, Master’s Thesis, Dept. of Electrical and Computer Engineering, Virginia Tech, Blacksburg, Spring 2000.
7. <http://kbs.cs.tu-berlin.de/~jutta/toast.html> (current November 2001)
8. A. Varma and V. Subramanian, *ECpE 5514 – Assignment #6*, Class Project, Dept. of Electrical and Computer Engineering, Virginia Tech, Blacksburg, Fall 2000.
9. Panda, P.R. “SystemC - a modeling platform supporting multiple design abstractions”, *Proceedings of the 14th International Symposium on System Synthesis*, 2001, pp 75 –80
10. Synopsys Inc, Coware Inc, and Frontier Design Inc, *SystemC Version 1.1 User’s Guide*, 2000
11. SpecC Technology Open consortium, “SpecC – Objective”
<http://www.specc.gr.jp/eng/purpose/index.htm> (current November 2001)

12. K. Kranen, "SystemC Forum – Re: SpecC and SystemC",
<http://www.systemc.org/hypermail/systemc-forum/0054.html> (current November 2001)
13. R. Doemer, "SpecC Reference Compiler",
<http://www.cecs.uci.edu/~specc/reference/> (current November 2001)
14. R. Goering, "SpecC Consortium Gloats at SystemC Turmoil",
<http://www.eet.com/story/OEG20010606S0074> (current November 2001)
15. Forte Design Systems, "Forte Design Systems – Cynlib",
<http://www.forteds.com/products/cynlib.html> (current November 2001)
16. Forte Design Systems, "Forte Design Systems – Extended SystemC Cynlib (ESC)",
<http://www.forteds.com/esc/index.html> (current November 2001)
17. M. Santarini, "Cynlib Joins OCSI, Merging System-level Languages",
<http://www.eetimes.com/story/OEG20011113S0058> (current November 2001)
18. Forte Design Systems, "Forte Design Systems Joins the Open SystemC Initiative",
http://www.forteds.com/esc/press_release.html (current November 2001)
19. Co-Design Automation, Inc, "Superlog from Co-Design Automation",
<http://www.superlog.org/> (current November 2001)
20. Co-Design Automation, Inc, "The Co-Design Product line",
<http://www.co-design.com/product/index.htm> (current November 2001)
21. Open SystemC Initiative, "Products & Solutions",
<http://www.systemc.org/products.html> (current November 2001)
22. J. R. Armstrong, "ECE 5514 – Modeling with Hardware Description Languages",
<http://mit.iddl.vt.edu/courses/ece5514/sectiona/> (current November 2001)
23. CoWare Inc, "CoWare N2C Datasheet",
<http://www.coware.com/cowareN2C.html> (current November 2001)
24. http://www.gsmworld.com/membership/ass_sub_stats.html (current November 2001)
25. J. G. Proakis, Digital Communications, Third Edition, McGraw-Hill International Editions, Singapore, 1995.

26. J. Degener, "Digital Speech Compression",
<http://www.ddj.com/documents/s=1012/ddj9412b/9412b.htm> (current November 2001)
27. http://gsm.csie.nctu.edu.tw/course/ppt/speech_coding.ppt (current November 2001)
28. Y. H. Hu, "Introduction to Speech Coding",
<http://www.cae.wisc.edu/~ece734/notes00/SpeechCoding.ppt> (current November 2001)
29. D. Deng, "Vocoders", Advanced Digital Signal Processing Class Lecture,
<http://www.ee.latrobe.edu.au/~dennis/teaching/ELE41ASP/LECT20.PPT> (current November 2001)
30. J. M. Huerta and R. M. Stern, "Speech Recognition from GSM Codec Parameters", http://www-2.cs.cmu.edu/afs/cs/user/robust/www/Papers/icslp98_juan.pdf (current November 2001)
31. S. M. Redl, M. K. Weber and M. W. Oliphant, *An Introduction to GSM*, Artech House Publishers, Norwood, 1995.
32. L. Green, "A5/1 Pedagogical Implementation", <http://jya.com/a51-pi.htm> (current November 2001)
33. "Crack A5", <http://crypto.radiusnet.net/archive/cryptanalysis/crack-a5.htm> (current November 2001)
34. B. Schneier, *Applied Cryptography*, Second Edition, John Wiley & Sons Inc, New York, 1996
35. R. Jigorea, S. Manolache, P. Eles, and Z. Peng, "Modelling of real-time embedded systems in an object-oriented design environment with UML", *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*, 2000, pp 210 - 213
36. G. Fayad, and K. Khordoc, "An object-oriented refinement methodology through the design of a settop-box", *Canadian Conference on Electrical and Computer Engineering*, 2000 Volume: 2, pp 1032 -1036
37. J. R. Armstrong and Y. Ronen, *Modeling with SystemC: A Case Study*, 2000

38. G. Economakos, P. Oikonomakos, I. Panagopoulos, I. Poulakis, and G. Papakonstantinou, "Behavioral Synthesis with SystemC", *Proceedings of Design, Automation and Test in Europe*, 2001, pp 21-25
39. Synopsys Inc., *CoCentric SystemC Compiler Behavioral Modeling Guide, Version 2000.11-SCC1*, December 2000

Vita

Anup Varma was born on September 15, 1975 in Trivandrum, the capital city of the state of Kerala in India. He graduated with a Bachelor of Technology degree in Applied Electronics & Instrumentation Engineering from the University of Kerala in 1997. After working in the software industry for more than two years, he decided to pursue his higher studies in the field of ASIC design. He graduated with a Master of Science degree in Electrical Engineering from Virginia Tech in 2001.