Software Synthesis of SystemC Models

Brijesh Sirpatil

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

July 2002
Blacksburg, Virginia

Keywords: SystemC, Software Synthesis, Embedded software, GSM

Abstract

Software Synthesis of SystemC Models.

Brijesh Sirpatil

Technological advances are providing us with the capability to integrate more and more functionality into a single chip. This is leading to a new design paradigm, System On a Chip (SOC). In SOC designs all the functionality of a system is put inside a single chip, leading to increased performance, reduced power consumption, lower costs, and reduced size. SOC design brings with it new challenges and difficulties, however. The designs are now large, complicated and involve both software and hardware components. The designs have to be modeled at a high level of abstraction before partitioning into hardware and software components for final implementation.

SystemC is a system level modeling language useful for System On a Chip design. It provides various features to perform system level modeling and simulation, which are missing in the generic HDL's such as VHDL and Verilog. The hardware portion of the SystemC models can be synthesized into hardware using commercial tools . The software portion can be rewritten as embedded software for the target processor.

The aim of this thesis is to explore the SOC design process and to define methods for software synthesis of SystemC models. Software synthesis involves translation of SystemC models into code that is suitable for execution on an embedded processor. A simple scheduler that replaces the SystemC simulation kernel is proposed. This scheduler allows SystemC models to be executed directly as embedded software without the need for extensive modification or translation. Application of this process to the development of a GSM speech processing system, including the translation of part of the SystemC model into software that will execute on an embedded processor, is shown and the results are presented.

Table of contents

1 Introduction 1
1.1 SOC Design Paradigm 2
1.2 SOC Design Issues 4
1.3 Modeling tools for SOC design paradigm 4
1.4 Aim of thesis 5
1.5 Overview of Thesis 6
2 SystemC Language 7
2.1 SystemC Language Features 7
2.1.1 Modules and processes 7
2.1.2 Ports and Signals 7
2.1.3 Data Types 8
2.2 SystemC Simulation Kernel 8
3 GSM Speech Processing 11
3.1 Speech Encoder 12
3.2 Channel Encoding 12
3.3 Interleaving 12
3.4 Encryption 13
3.5 Packet Formatting 13
3.6 Differential Encoder 13
3.7 Transmission 14
4 SystemC Model of GSM Speech Processing 15
4.1 Module Architecture 16
4.2 Handshake Signals 17
5 Embedded Processor 21
5.1 Computational Load of the Modules 21
6 Software Synthesis 23
6.1 Scheduler 25
6.2 Software Implementation of Ports and Signals 25
6.3 Software Implementation of Clocked Threads 27
6.4 GSM Model 29
6.5 Modeling Guidelines 32
6.6 Suggested Organization 32
7 Results 35
8 Conclusion 38
9 References 39
10 Appendix 40

List of Figures

Figure 1 Increasing system complexity. 1
Figure 2 Typical components of SOC design 2
Figure 3 SystemC simulation cycle. 9
Figure 4 SystemC simulation flow. 10
Figure 5 GSM speech processing. 11
Figure 6 Speech packet interleaving. 13
Figure 7 Speech packet format. 13
Figure 8 SystemC model of GSM speech processing. 15
Figure 9 Module architecture 16
Figure 10 Module architecture 17
Figure 11 Inheritance diagram for the module organization. 23
Figure 12 Scheduler for the software implementation 24
Figure 13 Handshake signals in software implementation. 29
Figure 14 Handshake process on sending side 29
Figure 15 Handshake process on the receiving side. 30
Figure 16 Current Architecture of the SystemC model of GSM speech processing. 33
Figure 17 Hardware/software compatible module architecture. 34
Figure 18 Model implementation flow. 34

List of tables.

Table 1 Execution time of the modules running on embedded processor. 22
Table 2 Execution times for the pure software implementation. 35
Table 3 Comparison of bit array and word array transfer models execution times. 36
Table 4 Comparison of pure software implementation and SystemC derived implementation 37

1 Introduction

In recent years there have been rapid technological advances in the semiconductor industry. Continuing advances in IC fabrication technology and material science have made it possible to keep up with Moore's Law [19]. The number of transistors on a chip and the clock frequency have been doubling every 18 months. This has made it possible to design complex systems within a single chip, leading to new architectures and design paradigms.

In the past, systems were built using discrete components such as microprocessors, memory and analog components. These systems do not scale well, in terms of complexity, performance, speed and cost. To increase the performance beyond that possible with discrete components, one has to integrate functionality into a single chip. The need for integration of functionality gave rise to VLSI designs. A single VLSI chip usually implements a complete sub-system or a large part of the needed functionality. A typical system today includes various VLSI cores, memory, microprocessors and the embedded software running on the processors. Total system complexity now includes the complexity in the silicon cores and the embedded software. Figure 1 shows the growth of system complexity with time.

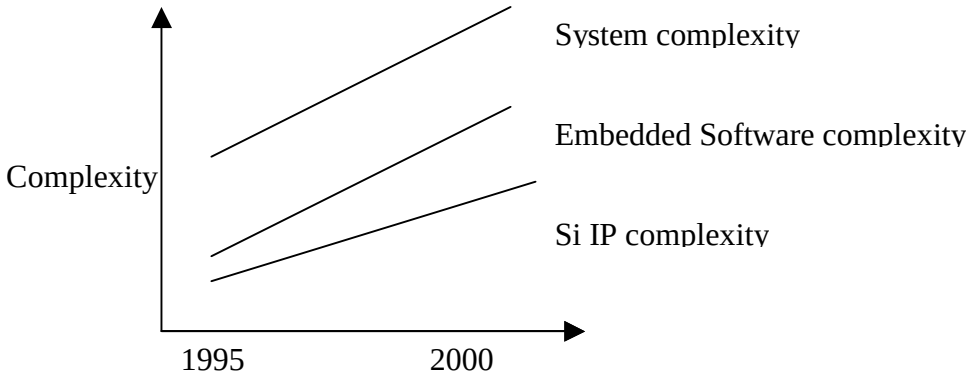


Figure 1 Increasing system complexity.

Increasing demands for more performance have taken the system designs based on VLSI chips to their limits. Now the basic gate delay is no longer the speed/performance bottleneck. The bottleneck now is the interconnect delays, power consumption and low system bus speeds. One way to overcome the above bottlenecks is to put all the various VLSI cores, memory, and processors into a single chip. This eliminates latency and delays of accessing data external to the chip, thereby increasing the performance. The tendency to put more functionality into a single chip has led to large and complex designs. The older design flow and methodology cannot cope up with the increased complexity. In the early stages of the design, not only the hardware, but also the entire system including the software has to be modeled to verify and validate the design.Engineers have begun to use a new design paradigm, System On a Chip (SOC), to overcome the the above mentioned challenges.

In the SOC design paradigm, all the functionality of a complete system is put into a single silicon die. The usual SOC chip may consist of a microprocessor, memory, glue logic, peripheral devices and analog modules (Figure 2). The SOC design paradigm enables reuse of silicon IP cores. Designers can now build complete systems by putting together various IP cores inside a single chip. This leads to reduced development time and costs. Complete integration of all the functionality within a single chip means better performance, speed, lower power and higher reliability.

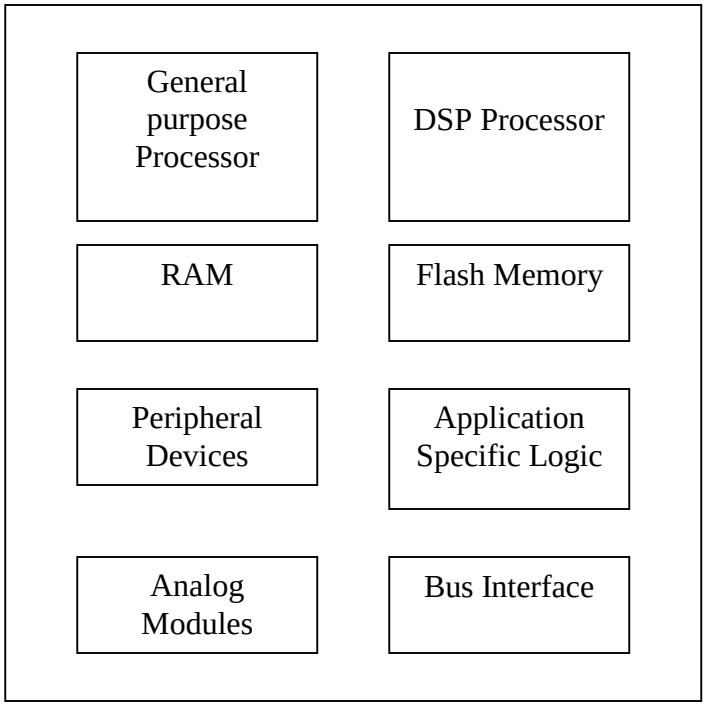


Figure 2 Typical components of SOC design

The SOC design paradigm is made possible with recent advances in IC fabrication technologies. With the capability to pack more and more transistors into a single die, we are able to put more functionality into a single chip. This allows a designer to pack all the functionality of a product into a single chip, giving rise to SOC designs.

1.1 SOC Design Paradigm

A typical SOC design is a complex system with hardware and software components interacting with each other to perform a given task. As discussed above, the SOC may consist of ASIC cores, peripherals, and a general processor with software . Various IP cores that are fully developed and tested by third-party sources may be included. For efficient implementation and reduced development time, it is important to have an early and accurate high-level model of the entire system. A designer needs to explore the architecture, develop software, integrate systems and measure system performance before

the hardware is built. Based on the performance of the model, the designer can then partition the system into hardware and software components and study the trade offs of a given partition.

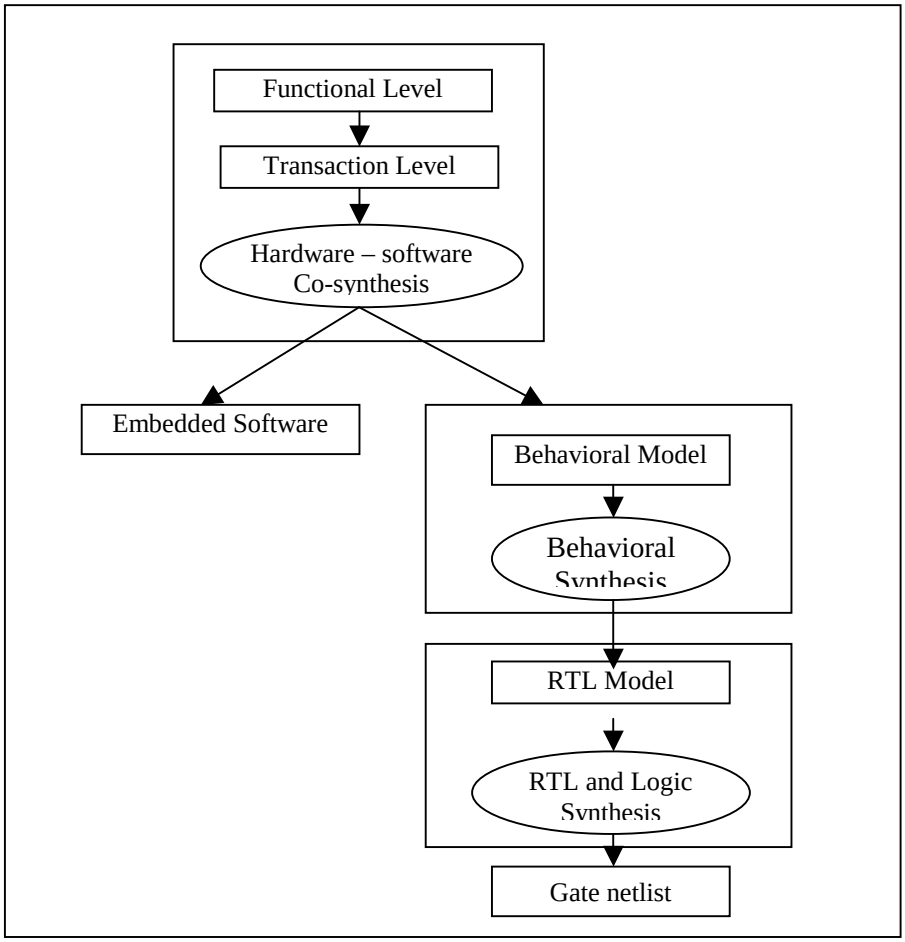


Figure 3 Typical SOC design flow.

A typical design flow of an SOC system is described in [1] and is shown in Figure . The system is first modeled at the functional level or transaction level. The functional level model is an un-timed model and composed of function calls. The transaction level model is a timed model, and interactions between models are through signals and events. At this level of modeling, the architecture and algorithms are verified. Any performance issues and bottlenecks are studied and simulated. Once the architecture and algorithms are verified, the next step is to determine which part of the system is to be implemented in hardware and which part goes into software. This process is called hardware/software partitioning. The software portion runs as embedded software on the general-purpose microprocessor and the hardware portion is implemented as an embedded ASIC core.

To partition the system, the computational complexity and implementation cost of each of the sub-systems is measured or estimated. These values are then used to arrive at a hardware/software partition that meets all the requirements in terms of timing requirements, development and production costs, development time, and die area. The usual measure of the cost of a software implementation is the computational load and timing restrictions on the embedded software. For the hardware implementation, cost is measured by die area (number of gates) and cost of production. Once a suitable partition is obtained, the hardware subsystem may have to be re-written in a suitable HDL so as to be compatible with the synthesis tools. The software part of the system would have to be developed for the embedded processor. This transition from a high level of abstraction to a lower level of abstraction is usually done manually.

1.2 SOC Design Issues

The development cycle of a complex SOC design involves modeling and testing of the system at various levels of abstractions. The process of converting from one level of abstraction to another is time-consuming and laborious. Added to that, at every step of the transition between models, one needs to simulate and verify the design. This testing and verification is again an expensive and time-consuming process. Often, one may have to re-write the test benches if there is a shift in modeling platform.
A single modeling language that can used to describe a system at all levels of abstraction would considerably reduce design time and effort. The need to rewrite the model during design flow would be eliminated. The same test benches could be used at all the levels of abstraction, leading to reduced costs and development time. Using a single language would also ensure that the models are consistent and error-free across all levels of abstraction. Thus, there is a need for a modeling language/platform that can scale effectively from high-level behavioral modeling to low-level abstraction of RTL models.

The modeling platform should also support synthesis of the models into either hardware or software components. An SOC modeling platform has to have native synthesis tools, as conversion of models from one platform to another is an expensive process. Just like there are tools for hardware synthesis, there is a need for tools to synthesize software. There are tools that convert high-level abstract models into a hardware circuit, but similar tools for software synthesis are non-existent. To manage the ever-growing complexity of systems, the automation of software synthesis steps will no longer be an option but a necessity. In the following sections we will exam the current state of tools available for SOC designs.

1.3 Modeling tools for SOC design paradigm

- VHDL and Verilog are the two most popular and widely used hardware description languages. They are well suited for modeling hardware, and the accompanying synthesis tools are mature and produce optimized hardware. But, the drawback is that neither language has suitable constructs for high-level system modeling. They also do not support hardware-software co-modeling and co-simulation, and they are very poor in modeling software constructs. Other limitations of VHDL and Verilog include poor simulation speed and efficiency,

and the inability to incorporate existing C/C++ IP which has been tested, debugged, and optimized into designs.

There is a need for a modeling language that can scale from high-level abstract modeling to low-level RTL modeling. Some of the new languages that fall into this category are SystemC, Cynlib, and Superlog.

SystemC [2] is a C++ class library for modeling system level designs. SystemC is primarily targeted towards modeling of complex System On Chip (SOC) designs. It is an industry-sponsored open standard for system-level modeling platforms. Since SystemC is based on C++ classes, it inherently supports the modeling of software. It also has classes to model hardware constructs such as signals and ports. SystemC has a built in simulation kernel. A general purpose C++ compiler can used to compile the SystemC model. The output of the compiler is an executable file, which upon execution simulates the model. Models can be developed and debugged using general tools such Visual Studio or GNU's gcc/gdb. SystemC models can output trace files that are compatible with standard waveform display tools.

Cynlib is also based on a C++ class library [3]. It is a set of C++ classes which implement features necessary for modeling hardware. The library creates a C++ environment in which both the hardware and the test environment can be modeled and simulated. However, the focus of Cynlib is more towards hardware modeling in C++ rather than system-level modeling.

Superlog is an extension of Verilog with support for C language features. It is not compatible with general C/C++ compilers and needs its own set of tools for simulation.

From the above description of the languages, one can see that only SystemC is specifically targeted towards system-level modeling. Since it is based on C++ class libraries, it inherently supports all of the C++ language constructs. It can be compiled using a general C/C++ compiler for simulation. Synopsis offers a compiler tool [4] to synthesize the SystemC models into hardware. SystemC offers a seamless design flow from high-level modeling to RTL level modeling and final hardware synthesis. SystemC does lack tools for automated software synthesis. But, since the SystemC is based on C++, its models can be easily ported to run as embedded software. Hence, in today's market, it is a suitable candidate for hardware-software co-design and simulation.

1.4 Aim of thesis

A case study of using SystemC as a high-level modeling language is presented in [5]. The authors conclude that SystemC is well suited for such a task. Behavioral synthesis of SystemC models is presented in [6]. Modeling guidelines and a study of hardware compiler tools is presented in [7][8].

The aim of this thesis is to explore the process and to define methods for software synthesis of SystemC models. Software synthesis involves the translation of SystemC models into code that is suitable for execution on an embedded processor. The motivation behind such a translation is to eliminate the time consuming process of re-implementing

the models as embedded software. Some guidelines and restrictions for developing SystemC models that are easily synthesized into software are presented. A method for preserving the structure and semantics of SystemC models during the translation to software code is proposed, based on the use of a simple scheduler that replaces the SystemC simulation kernel . Application of this process to the design of a GSM communication system, translating part of the SystemC model into software that will execute on an embedded processor, is shown and the results presented. The work leading to this thesis was also published in paper [9].

1.5 Overview of Thesis

Chapter 2 describes in brief the features and modeling constructs of SystemC HDL. It also elaborates the simulation steps and flow of the SystemC simulation kernel.

Chapter 3 presents the details of GSM speech processing and transmission. All the steps involved in speech processing are explained in brief.

Chapter 4 presents the SystemC model of the GSM speech processing. It delves into architecture of the modules and handshake signals used between the modules.

Chapter 5 discusses the target embedded processor and reasons for its choice. It also presents the computational load of all the modules on the target processor.

Chapter 6 delves into details of software synthesis. It presents the idea of using a scheduler to schedule threads and gives the details of implementation of the scheduler. It also contains pseudo code and examples of using the scheduler and software signals. The chapter also presents modeling guidelines and coding restrictions for software synthesis.

Chapter 7 presents the results, performance and comparisons of the SystemC derived implementation of embedded software against pure software implementation.

Chapter 8 concludes the thesis and provides pointers to future work.

2 SystemC Language

SystemC is a C++ class library for modeling system-level designs[2]. SystemC is primarily targeted towards high-level modeling of complex systems. Using SystemC one can effectively create cycle accurate models of algorithms, hardware architectures, and the interfaces between them. Since SystemC is based on C++, it naturally supports software algorithm development. On the other hand, to model hardware, it provides necessary constructs for timing and concurrency. SystemC has a built in simulation kernel, so it does not require any tools for simulation. SystemC can be compiled using standard C++ tools to create an executable model that can be used for simulation and validation.

2.1 SystemC Language Features

Important SystemC modeling constructs are described below in brief.

2.1.1 Modules and processes

VHDL uses an *entity* and Verilog uses a *module* to encapsulate the logic and structure of hardware modules. Similarly SystemC has *module*, which encapsulates the data and algorithms. Modules in turn contain *processes*, *ports* and *signals.* A *process* is used to model concurrency and is the basic unit of simulation. *Processes* are sensitive to signals and are executed concurrently. There are three types of processes available for modeling – methods, threads, and clocked threads.

Methods: Methods are executed whenever an event occurs on a signal in the method's sensitivity list. Once the execution begins it cannot be suspended; it completes execution and returns control to the simulation kernel. Hence, a method may not contain an infinite loop.

Thread: Threads can be suspended and activated by the simulation kernel. A *wait()* function call suspends the thread. It is re-activated again whenever an event occurs on a signal in the thread's sensitivity list, and execution continues from the next statement. A thread can contain an infinite loop with at least one *wait()* function call.

Clocked Thread: Clocked threads are a special case of Threads sensitive only to the clock signal. Clocked threads are useful for hardware synthesis and current synthesis tools support only clocked thread processes.

2.1.2 Ports and Signals

Ports provide the external interfaces to modules and pass information between them. They are similar in function to VHDL and Verilog input/output ports. There are three types of ports – input, output and bi-directional ports, depending on the direction of data flow.

Just the way signals are used to interconnect ports in VHDL signals are also used in SystemC to interconnect ports. Signals transfer data from one port to another. Ports and Signals can be of any data type supported by SystemC.

When a port is read, the value of the signal the port is connected to is returned. When a port is written, the value of the signal the port is connected to is updated. When a port is written, the signal value is not updated immediately, however, but at the end of the simulation cycle. This ensures that all the processes see the same value of the signal within a simulation cycle.

2.1.3 Data Types

As SystemC is based on C++, it supports all the native data types of the C++ language, such as integer, float, and char. Pointers can be used in high-level models and for simulation, but cannot be synthesized with the current synthesis tools. SystemC also has some additional data types for modeling logic and hardware, such as sc_bit and sc_logic. Sc_bit is a 2-valued data type and sc_logic is a four valued (0,1,X,Z) data type. SystemC also has fixed-precision signed and unsigned integer data types where the user can specify the number of bits used to represent a number. SystemC also provides signed and unsigned fixed-point data types that can be used to accurately model DSP systems.

2.2 SystemC Simulation Kernel

SystemC designs can be compiled using any ANSI C++ compiler. SystemC has a built in cycle-based simulation kernel to simulate the designs. The resulting executable specification realizes the model and the simulation kernel. The complete simulation kernel is built into the class library and needs no external tools for simulation of the model. The source code for the kernel and the library is available with the distribution of the SystemC platform, from [2]. Wolfgang Muller, et al, have published a rigorous description and semantics of the SystemC simulation kernel [16].

Each one of the user-defined processes is executed independently of the others and also the kernel. Simulation begins with a call to the function sc_start(). At the start of the simulation all the processes are initialized and scheduled for execution. All of the processes get a chance to execute in every simulation cycle. The order of execution is not defined. Any changes in the signal values are not immediately updated. Signals are assigned new values only in the next simulation cycle. This makes the simulation cycle accurate. A process that is executing or is scheduled to be executed is in an active state. An active process goes into a suspended state after it completes its operation or reaches a wait statement. Once all the processes are in a suspended state, the kernel then updates the signals, advances simulation time and enters into the next simulation cycle. The simulation cycle is illustrated in Figure 3 (adopted from [16]).

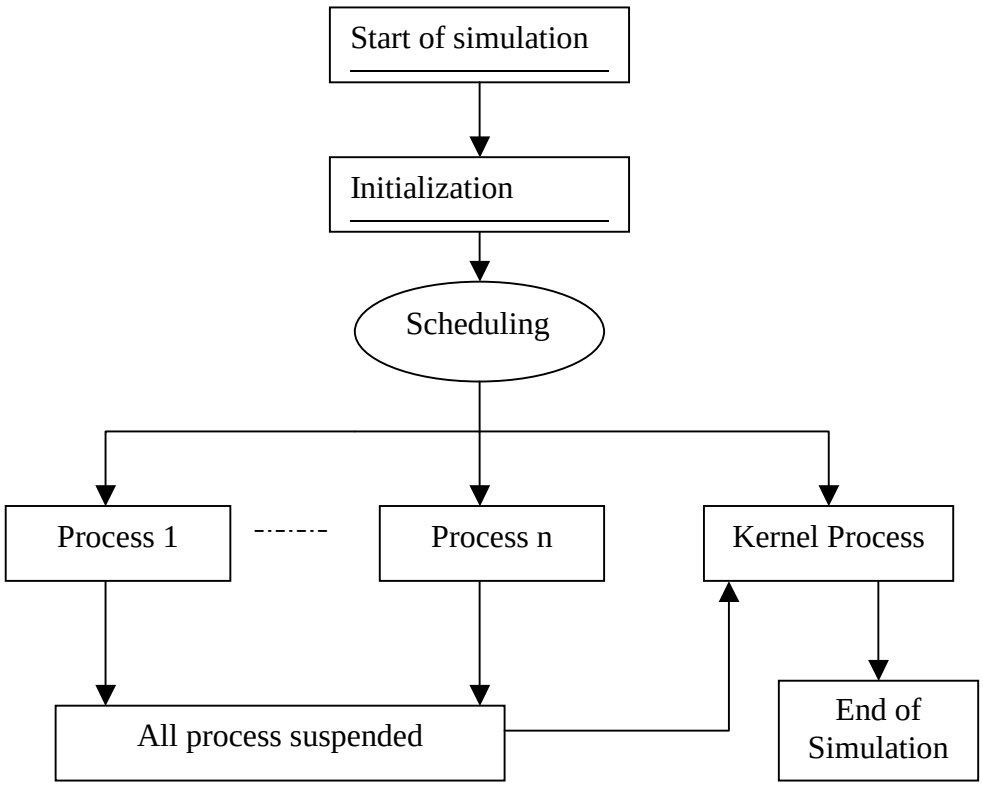


Figure 3 SystemC simulation cycle.

At the start of the simulation, the module initialization or the test bench generates the initial events. These events then trigger processes (Figure 4). Any processes that were activated are then executed. Clocked threads, referred to as Cthreads, are sensitive only to the clock signal and are scheduled to be executed in the future. Once all the processes have been executed, then the signals are updated. The updating of signals may cause new events, which may trigger other processes. The triggered processes are then executed, which may in turn trigger other processes. This cycle continues until there are no events triggering any of the processes or all the processes have been executed. Once all the processes are in the suspended state and there are no events, then the CThreads are executed. After execution of the Cthreads, simulation time is advanced and the clock and all the signals are updated. This completes one simulation cycle. This cycle is then repeated until simulation comes to an end or is stopped.

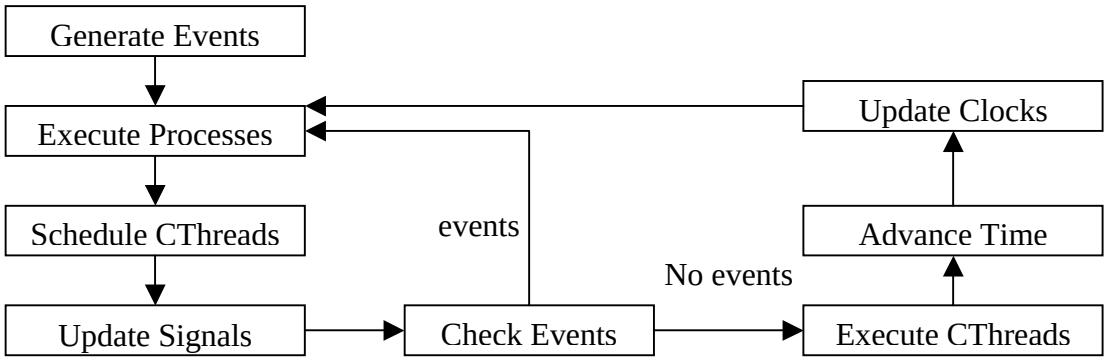


Figure 4 SystemC simulation flow.

SystemC is based on a C++ class library; therefore, theoretically it is possible to port the SystemC library to any embedded processor. By doing so, there would be no need for software synthesis. However, this step is neither feasible nor practical. The SystemC kernel carries with it a large overhead and performance penalty, which would be unacceptable in embedded applications. Since the kernel is designed for cycle-accurate simulation, it has large latency and will not meet the strict timing requirements of embedded systems. Also, the SystemC library is currently available only on Windows, Solaris and Linux OS platforms. The library depends on an operating system to provide certain functionalities. To execute a SystemC model on an embedded system would require the embedded system have an OS. The OS comes with its own overhead in terms of memory and computational load, which again may not be acceptable in some embedded applications. Hence, it is not viable to simply port the complete SystemC library and simulation kernel over to the embedded processor. One needs to be able to execute the SystemC models without the overhead of the cycle accurate simulation kernel.

3 GSM Speech Processing

To effectively study the software synthesis process and to come up with process, method and design guidelines, we need a complex real world system. The system must have modules, which can be modeled as processes. The modules should have interactions among themselves and affect behavior of each other. Finally, the computational load should be large enough that we would have to partition the system into hardware and software for optimum performance.

The Global System for Mobile telecommunications (GSM) is a digital cellular communications standard [17][18]. It was originally developed in Europe to create a common European mobile telephone standard, but it has been rapidly accepted worldwide. GSM speech processing is a complex and computationally heavy system. It consists of various well-defined processing steps, some of which are mathematically intensive and operate on integer values. Other processing steps are algorithmically complex and process data in bits. Hence, we find that GSM speech processing is an ideal candidate for our work.

The steps involved in GSM speech processing and transmission are illustrated in the figure below (Figure 5). Each of the steps involved is briefly explained in the following paragraphs.

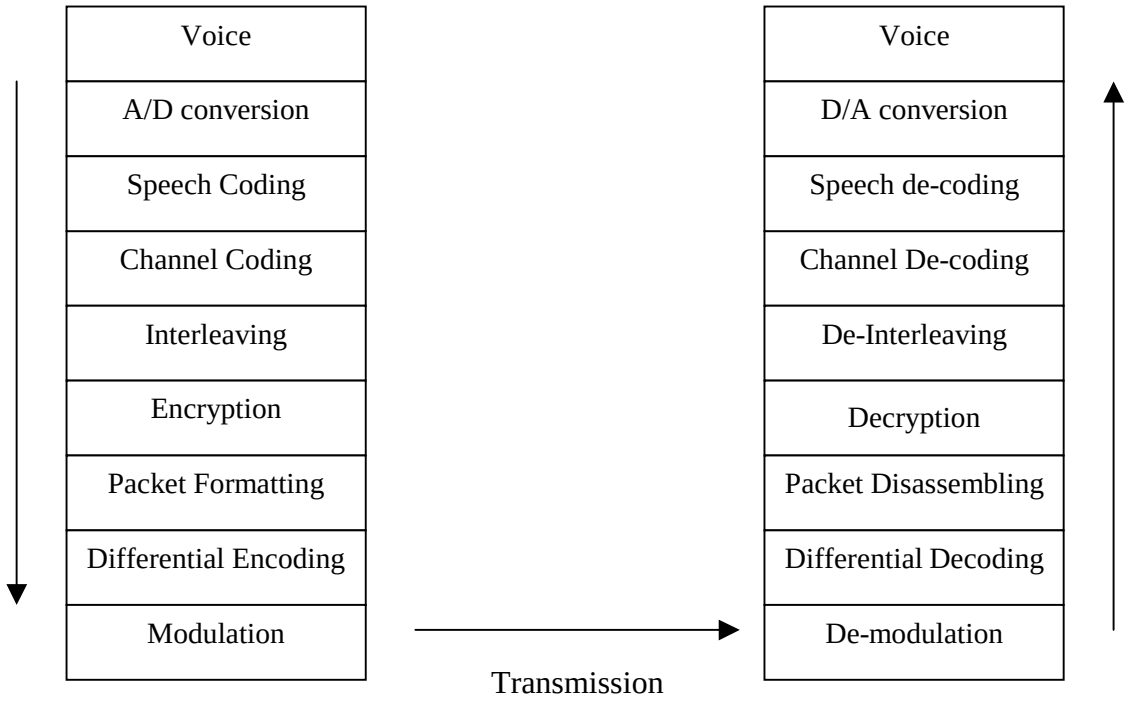


Figure 5 GSM speech processing.

3.1 Speech Encoder

The speech codec used in GSM is RPE-LTP (Regular Pulse Excitation-Long Term Prediction). The codec models the human vocal tract using two filters and an initial excitation. It transmits the parameters necessary to model the vocal tract and to recreate the speech at the other end. The speech encoder takes in 20ms of speech as input. Speech is sampled at 8 KHz giving total of 160 signed 13 bit PCM samples in each 20ms segment. The encoder then compresses the 160 samples into one frame of 260 bits. The speech encoder outputs data at the rate of 13kbps (260bits / 20ms).

3.2 Channel Encoding

Channel coding is performed to detect and, if possible, correct errors that occurred during the transmission. It adds redundancy bits to the original information in order to detect and correct errors. GSM uses both a block code (parity encoding) and a convolutional code. The coding differs for the data, speech and control channels. Since we are only modeling the speech channel of the GSM system, speech channel encoding is described in the following paragraph. More information regarding channel coding can found in [14].

The 260 bits of a GSM speech frame are divided into three different classes according to their function and importance. The most important class is the class Ia, containing 50 bits. Next in importance is the class Ib, which contains 132 bits. The least important is the class II, which contains the remaining 78 bits. The different classes are coded differently. First of all, the class Ia bits are block-coded (parity encoding). Three parity bits, used for error detection, are added to the 50 class Ia bits. The resultant 53 bits are added to the class Ib bits. Four zero bits are added to this block of 185 bits (50+3+132). A convolutional code, with r = 1/2 and K = 5, is then applied, obtaining an output block of 378 bits. The class II bits are then added, without any protection. An output block of 456 bits is finally obtained.

3.3 Interleaving

Interleaving is used to obtain time diversity in a digital communications system without adding any overhead. The interleaving decreases the possibility of losing whole bursts during the transmission. The interleaving scheme used for the speech channel is described in the following paragraph.

The total of 456 bits from the convolutional encoder, which constitutes 20ms of speech, is subdivided into eight blocks of 57 bits each. These eight blocks are then transmitted in consecutive time slots. If one of the blocks is lost due to burst errors, the other 7 blocks would contain enough information so that whole segment can be recovered using error correction. Each time slot carries two 57-bit sub-blocks of data from two different 20ms speech segments. This is illustrated in the figure below (Figure 6).

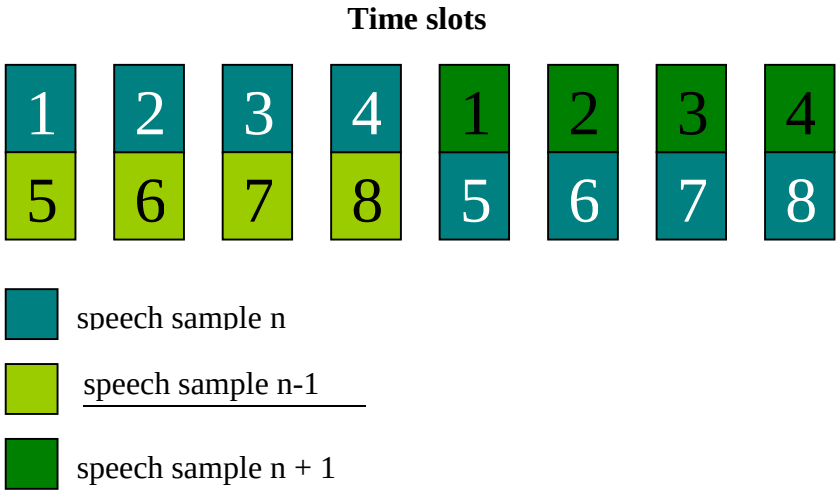


Figure 6 Speech packet interleaving.

3.4 Encryption

To provide privacy and prevent unauthorized network access, the eight blocks of interleaved data are encrypted before burst formatting and transmission. Two types of ciphering algorithms are used in GSM, which are referred to as the A3 and A5 algorithms. These algorithms are not published for security reasons. For our work, we needed the computational load and complexity, but not the algorithmic details. Using some information from the Internet [10][11] and textbooks on algorithms [12], Anup Varma [8] implemented an approximation of the algorithms. This implementation simulates the computational load of encrypting and decrypting the data, which is sufficient for our work.

3.5 Packet Formatting

The encrypted data is placed into a packet (also referred to as a frame), which contains additional information for synchronization, equalization and control signals. The structure of the packet is shown below (Figure 7).

3 Start Bits	57 bits of speech data	1 stealing flag	26 training Bits	1 stealing flag	57 bits of speech data	3 Stop Bits	8.25 guard bits

Figure 7 Speech packet format.

3.6 Differential Encoder

To demodulate a transmitted signal, a receiver needs to be synchronized with the transmitter's clock or carrier wave. This is usually accomplished by transmitting the carrier signal along with the modulated signal. Before the packet is transmitted, the binary stream is differentially encoded. Differential encoding of data removes the need for transmitting the carrier, as the data is encoded not in the phase of the carrier but in the

phase changes. The differential encoder output is the XNOR of the present bit and the past bit.

3.7 Transmission

Once the bit stream is differentially encoded, it is ready for transmission. The modulation scheme used by GSM is Gaussian Minimal Shift Keying (GMSK). GMSK is a type of digital FM modulation, where the modulated signal is passed through a Gaussian filter to smooth the rapid changes in frequency. Rapid changes in frequency would tend to spread the energy of the modulated signal, thereby increasing the bandwidth. Therefore, passing the signal through a filter minimizes the bandwidth.

GSM uses two bands of 25 MHz, for transmission and reception.

- 890-915 MHz band is used for subscriber-to-base transmissions
- 935-960 MHz band is used for base-to-subscriber transmissions.

4 SystemC Model of GSM Speech Processing

A detailed description and tutorial of SystemC modeling is available in [13]. Anup Varma has developed a SystemC model of the GSM speech processing for his master's thesis [8].

Speech is processed in 20ms segments. Data flow is linear from the first stage to the last stage. Within stages, however, there are some feedback loops and buffering is needed. The packet size varies as the data moves from one stage to another. Since any of the stages could be implemented in hardware or software, the interface between the stages had to be standardized. All the modules had a well-defined interface and architecture.

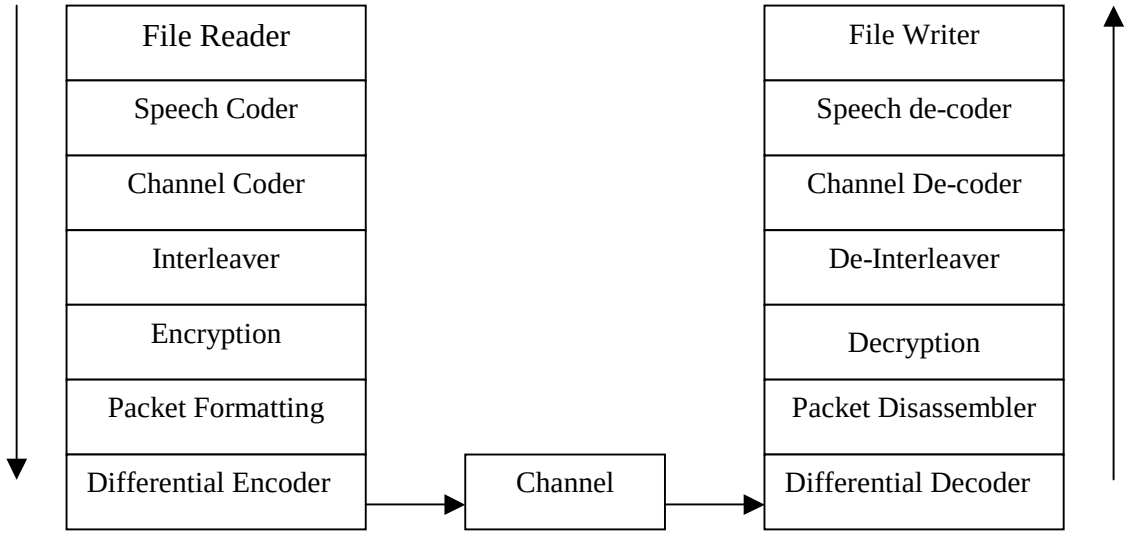


Figure 8 SystemC model of GSM speech processing.

Figure 8 shows the various modules in the SystemC model and the data flow among the modules. The file reader module acts as a data source for the speech encoder. It reads in 20ms of speech data and transfers it to the speech encoder. The speech encoder processes the data and transfers it down the chain to the next module, the channel encoder. The data is processed and moves down the chain from the channel encoder to the interleaver, the encryption module, the packet-formatting module, and finally, the differential encoder module. In the real systems, the output of the differential encoder goes to a modulator where it is modulated using the RF carrier frequency for transmission. In the SystemC model, the output of the differential encoder is fed into a channel module. The channel module adds random bit and burst errors to the bit stream, simulating the errors in signal transmission and reception.

On the receiving side, the differential decoder gets the bit stream from the channel module. This bit stream contains the random errors introduced by the channel. The differential module processes the data and moves it up the chain to the packet

disassembler. Data moves up the chain from the packet disassembler to the decryption module, the channel decoder and the speech decoder. The output of the speech decoder is an audio stream. The file writer module accepts the audio stream and writes it to a file for later playback.

4.1 Module Architecture

The main data flow in GSM speech processing is linear. Each module has to get data from the previous module, process the data, and then provide data to the next module in the chain. To make the models compatible with hardware/software partitioning, the core data processing and the data input/output functions were separated and implemented in separate sub-modules, as illustrated in Figure 9. All of the modules operate synchronously to a global clock. All the data transfer and signals are also synchronous to the clock.

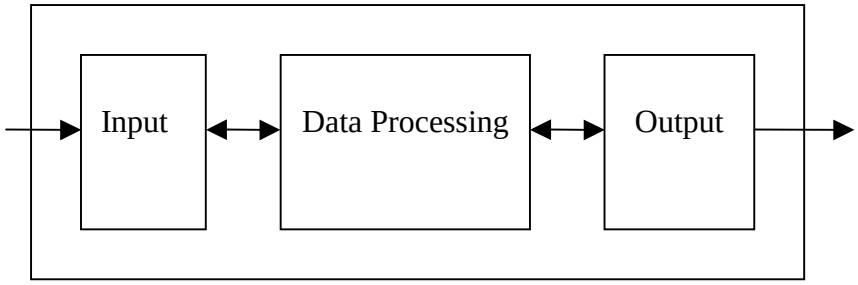


Figure 9 Module architecture.

The input, output, and processing sub-sections are implemented in separate processes within a module. This allows for concurrent execution of the subsections, leading to optimized performance. The processes communicate with each other using signals. The input sub-section writes the input data into an input buffer. The data processing sub-section operates on the data in the input buffer and writes the output into an output buffer. The output sub-section reads the data from the output buffer and transfers it to the next module.

4.2 Handshake Signals

A simple handshake protocol ensures reliable data transfer between the modules. The handshake protocol signals are described below. (Figure 10)

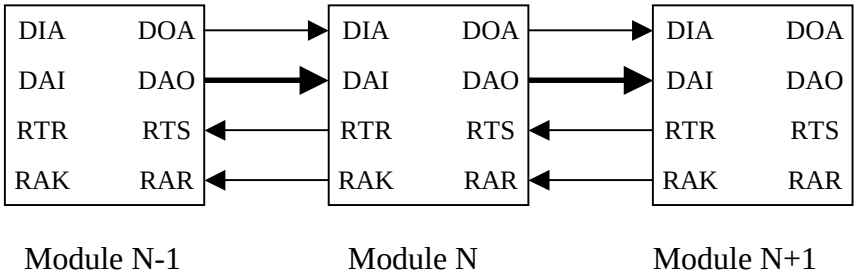


Figure 10 Module architecture.

- DOA (DataOut_Available): Output signal. Data is available for the next module. Asserted by the sending module.
- RTR (Ready_To_Receive): Output signal from the receiving module. Indicates module is ready to receive data.
- DIA (DataIn_Available): Input signal. Data is available to be received, asserted by the sender.
- RAK (Receive_AcKnowledge): Output signal. Acknowledgement from the receiver.
- RTS (Request_To_Send): Input signal. Receiving module is ready to receive data
- RAR (Recieve_Ack_Received): Input Signal. The acknowledgement from the receiving module indicating that it received data.
- DAI (Data In): Input data to the module.
- DAO (Data Out): Output data from the module.

Once a module is ready to receive more data, it asserts the RTR signal. The receiver module then waits until the DIA signal is asserted and begins to read the data in. The receiver acknowledges each data transfer across the bus (DAI) by asserting the RAK signal.

On the sending side, the sender waits until the receiver asserts the RTS signal. Once it sees RTS asserted, the sender asserts the DOA signal and writes out the data onto the DAO bus. The sender then waits for the acknowledgement RAR before writing the next data on to the bus.

The code for an input process is shown below. The input process is the same for all of the modules as it is a well-defined common interface. The data transfer is synchronous with handshake signals for acknowledgment. Data is transferred using a bus and the width of the bus is 16bits. This code snippet only shows the synchronization and handshake sections of the code. The code is taken from the interleaver encoder module.

```
void inter_encoder::input()
{
 input_reset();
 wait();
 while(true)
 {
        wait();

         // read input data from the bus
        for(int i=0;i<IE_MEMORY_SIZE;++i)
        {
          wait();
         //ready to accept the next word from the bus.
          I_GOT_YOUR_BIT.write(false);    //signal RAK
          READY_TO_RECV.write(true);     //signal RTR

        //wait till data is written to the bus
          wait_until(DATAIN_AVAIL.delayed() == true);    //Signal DIA
          word_input_data[i] = DATAIN.read();                 //read from bus DAI

          wait();
        //acknowledge the data
          I_GOT_YOUR_BIT.write(true);               //signal RAK
          READY_TO_RECV.write(false);               //signal RTR
          wait_until(DATAIN_AVAIL.delayed() == false);   //signal DIA
          wait();
         }
        wait();

        //complete data segment has been read from the previous module
…
      }
}
```

The code for the data processing process of a module is shown below. The code snippet shows only the handshake and synchronization sections. Again, the code is taken from the interleaver module.

```
void inter_encoder::process_data()
{
//process reset signal
process_data_reset();
 wait();
 while(true)
 {
```

```
        processing_started.write(false);
        wait();

        //wait till the input process has read the data segment
        wait_until(input_data_ready.delayed() == true) ;
        processing_started.write(true);

        //data processing code goes here

        wait();
        //indicate to the output process that data is ready
        input_data_processed.write(true);
        output_data_ready.write(true);
        input_ack_received.write(false);
        output_ack_received.write(false);

        wait();
        //wait for an ack from the output process.
        wait_until(input_ack.delayed() == true);
        input_data_processed.write(false);
        input_ack_received.write(true);

        wait_until(output_ack.delayed() == true);
        output_data_ready.write(false);
        output_ack_received.write(true);
        wait();
  }
}
```

The code for the output process is shown below. Again, only the handshake and synchronization sections are shown.

```
void inter_encoder::output()
{
  output_reset();
  wait();
  while(true)
  {
        wait();

        // send output data
        for(int i=0;i<IE_OUTPUT_SIZE;++i)
        {
                wait();

                //wait until receiver is ready
```

```
                wait_until(READY_TO_SEND.delayed() == true);  //signal RTS
                DATAOUT_AVAIL.write(true);                    //signal DOA
                DATAOUT.write(word_interleaved_data[i]);      //write to bus DAO
                wait();

                //wait for an ack.
                wait_until(YOU_GOT_MY_BIT.delayed() == true);//signal RAR
                DATAOUT_AVAIL.write(false);              //singal DOA
                wait();
        }
        wait();

        output_ack.write(true);
        wait_until(output_ack_received.delayed() == true);
        output_ack.write(false);
        wait();

    }
}
```

5 Embedded Processor

For our study we chose the StarCore SC140 processor [15] as the embedded processor in our SOC design. StarCore is an alliance between Motorola Semiconductor Products Sector and Agere Systems for the purpose of developing DSP core technology. The StarCore processor is targeted towards the communication market, and its architecture is well suited for mobile handsets. One of the most important considerations was that the StarCore is available as an IP core. Availability of StarCore DSP IP cores enables designers to build their SOC systems around the processor. We also had a development platform with a compiler and an instruction set simulator for the processor, which enabled us to compile and run our code to get timing measurements. For the above-mentioned reasons, the StarCore SC140 was chosen as our target embedded processor.

5.1 Computational Load of the Modules

To perform and study hardware-software partition tradeoffs, we need a measure of cost of implementation in hardware and software. In addition to the cost of implementation, we had to ensure that all the timing requirements were met. The measure of the cost of implementation in hardware was chosen to be the number of clock periods needed to perform the computation. The measure of the cost of implementation in software was chosen to be the number of processor clock cycles required to perform the computation. To simplify the calculations the hardware cost measurements were made at the same clock frequency as that of the processor.

To get the timing measurements on the embedded processor, each module was manually ported to run on the StarCore processor. Necessary changes in code were made to comply with the requirements of StarCore C compiler. Each module was run independently and the number of clock cycles required to process one block of data was recorded. The recorded values are shown below. The processor was running at 300 MHz.

Table 1 Execution time of the modules running on embedded processor.

Index	Module	M/c Cycles	Execution Time (ms)
1	A/D Converter		20.0000
2	Speech Encoder	1251510	4.1717
3	Parity encoder	12509	0.0417
4	Convolution. Encoder	71527	0.2384
5	Interleaving Encoder	105970	0.3532
6	Packet Encoder	6774	0.0226
7	A5 Encoder	43840	0.1461
8	Differential Encoder	9188	0.0306
9	Speech Decoder	488376	1.6279
10	Parity Decoder	12428	0.0414
11	Convolution Decoder	13387103	44.6237
12	Interleaving Decoder	97210	0.3240
13	Packet Decoder	4633	0.0154
14	A5 Decoder	42699	0.1423
15	Differential Decoder	8453	0.0282

6 Software Synthesis

If a C++ compiler is available for the embedded processor, then we could use the inheritance feature of the C++ language to arrive at an organization of the modules which lends itself to both hardware and software synthesis. We could encapsulate the core data processing in a base class. This base class would do all the data processing using synthesizable C language constructs. To simulate and synthesize it, we would derive the SystemC class from the base class. The SystemC class would provide all the necessary constructs for simulation and communication between modules. To implement it in software, we would derive a C++ class from the base class (Figure 11). This class would then take on the responsibility of creating threads, communication and synchronization with other modules and registering it with the scheduler.

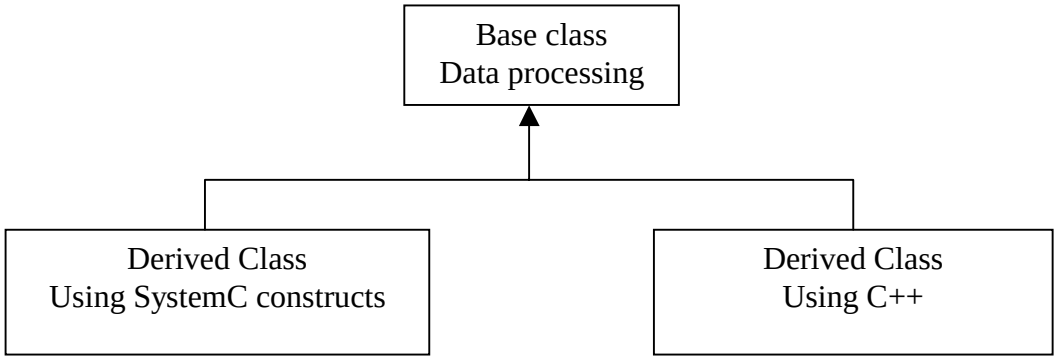


Figure 11 Inheritance diagram for the module organization.

Unfortunately, there is no C++ compiler for the chosen target embedded processor, the StarCore SC140. Infact there are very few C++ compilers for embedded processors. For this reason we had to come up with a different solution which only needs a C compiler. A software implementation that mimics the organization and architecture of the high-level SystemC model would be most easy and least time-consuming to implement as embedded software. Since SystemC is based on a C++ class library, it is possible to port the high-level model directly into software, including the simulation kernel. But, such a design would be very inefficient and would have the large overhead of the cycle accurate simulation kernel. Instead of porting the complete SystemC kernel, it is possible to execute SystemC models as software using a simplified scheduler. This scheduler can be easily implemented in C. A simplified scheduler would be lightweight and have much less overhead than the SystemC kernel.

Current day hardware synthesis tools for SystemC support only CThreads. Hence, any design that aims to be compatible with both hardware and software implementations has to use only CThreads. CThreads are processes that are sensitive only to clock signals. So, to execute a design based only on Cthreads, one needs a simple scheduler that schedules

all the active processes at every cycle. The simplified scheduler operation is illustrated in Figure 12.

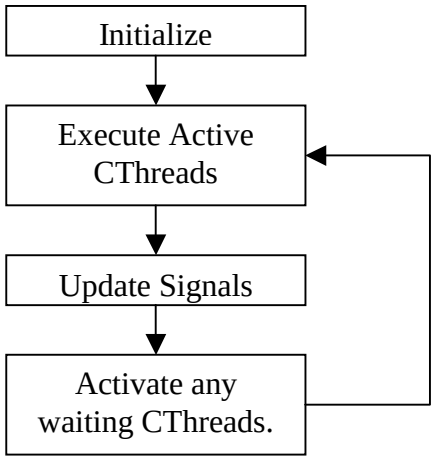


Figure 12 Scheduler for the software implementation

A simplified scheduler executes all the active threads. Any changes in the signal values are not immediately updated. Once all the threads in the present simulation cycle have had a chance to execute, the scheduler then updates the signals. This ensures that all the modules that read a certain signal see the same signal value in a given cycle. After updating the signals, the scheduler activates any threads that were waiting and are now ready to run. The scheduler then executes all the active threads, repeating the cycle.

The order of execution of the CThreads is not specified, which is also the case in a SystemC simulation. Once a CThread suspends, it is guaranteed that all the other CThreads get a chance to execute before the CThread is executed again. Since all the signal values are updated at the end of the cycle, this process ensures that all the CThreads read the same signal value in a given cycle. It also ensures that signal values are updated before the CThread is rescheduled. This leads to a direct analogy between the cycle of the scheduler and the hardware clock cycle of the SystemC simulation, although there is no concept of clock period and no guarantee that all the cycles take same amount of time to execute. This mechanism does ensure that any model that simulated correctly will execute correctly on the embedded software.

In summary, a scheduler that schedules CThread processes and updates signals can execute a SystemC model on the embedded processor. To implement SystemC models as embedded software, one has to implement a rudimentary scheduler along with support for software signals. Using this scheduler, one can port the SystemC models to embedded software with little or no modifications.

6.1 Scheduler

Craig Dry from Motorola has written and released a free scheduler, the Motorola 8101 Real-time Preemptive Scheduler (RPS). This scheduler formed the basis for our CThread scheduler. The original scheduler was extensively modified and extra features added to support threads and signals.

The scheduler initialization and thread creation functions are explained below.

`SchedInit(int stacksize)`
This function call initializes the scheduler. This should be called once at the beginning of the program. The argument `stacksize` is the size of the stack for the scheduler. The stack size is in bytes.
Example:

```
SchedInit(8000);
```

`SchedStart()`
A call to this function starts the scheduler. Before calling this function, the scheduler should be initialized and the threads created. Any mapping of ports and signals should also be completed. (Ports and signals explained in the next section).

6.2 Software Implementation of Ports and Signals

Modules in SystemC exchange data and control information using ports and signals connected to the ports. The port and signals were implemented as structures in the embedded software.

The signal structure is shown below.

```
//signal structure
typedef struct signal_struct {
    struct signal_struct *next;    //next signal in list
    int numBytes;        //size of signal type
    int updateFlag;      //whether signal has been
                         //updated or not
    void *current_val;   //current value of signal
    void *next_val;      //next value of signal
} Signal;
```

The scheduler stores all the signals in a linked list. The field `next` stores the pointer to the next signal in the linked list. The field `numBytes` defines the length of the signal in bytes. If one wants a 32-bit bus between two modules, then one has to create a signal with a length of 4 bytes. The signal structure stores both the current value of the signal and the next value of the signal. When a port connected to a signal is read, then the data pointed to by `current_val`, the current value of the signal, is returned. Whenever a signal is written to, the new value is stored in the location pointed to by `next_val.` Only when the scheduler updates the signal, is the new value copied into the current

value location. Since signals are all updated at the end of a cycle after all the active threads have been executed, all the threads see the same value of the signal during a simulation cycle. This ensures that model will work correctly without specifying any order of execution of the threads.

The `updateFlag` is used to optimize the process of updating the signals. Only those values that have been written in the present cycle will have the flag set. If the flag is set then the signal values are updated by copying the next value into current value.

To create a signal, one has to call the `CreateSignal()` function with the size of the signal in bytes. The smallest signal that can be created is one byte. As this is not hardware simulation, there is no overhead associated with the extra bits. The function returns a pointer to the signal structure.

Example:

```
pointer_to_signal = CreateSignal(size);
```

The port structure contains a pointer to the signal to which it is connected. The same port type is used for both input and output.

```
typedef struct port_struct {
    Signal *signal;           // signal connected to port
} Port;
```

A port has to be connected to a signal before it can be read or written. A code sample to connect a port to a signal is shown below.

```
//declare a Port and signal.
Port portA;
Signal *sigA;

//Create a signal with length of 1 Byte.
sigA = CreateSignal(1);

// connect the signals to the ports
ConnectPortToSignal(&portA, sigA);
```

Once the port is connected to a signal, it can be read and written. To read a port, the function `portRead()`is called. The function accepts two arguments; one is a pointer to the port. The other parameter is a pointer to the location where the read value is to be stored. Care should be taken that enough memory has been allocated to hold the complete signal.

```
portRead( struct port, char* ptr);
```

Code example:

```
//read portA and store the read value into location pointed
//to by data.
portRead(&portA, data);
```

To write to a port, the function portWrite() is called. The function accepts two arguments, a pointer to the port and a pointer to data that is to be written to the port. The size of the data to be written to the port should match the size of the signal connected to the port.

```
portWrite( struct port, char* data);
```

Code example:

```
//Write data present at the location pointed to by the data
into the port portA.
portWrite(&portA, data);
```

6.3 Software Implementation of Clocked Threads

In the SystemC specification, clocked threads execute independently and concurrently. To get the independent and concurrent execution in software, each clocked thread has to be implemented as a thread. One has to create a thread for every clocked thread process in SystemC and connect the modules using software signals. Any communication between the threads has to be through the use of signals.

To create and register a thread with the scheduler, the function call `createThread()` has to be called.

```
extern void createThread(int stackSize,void(*entryPoint)()
)
```

This function call registers a new thread with the scheduler. It allocates memory space for the stack used by the thread. The amount of memory is determined by the first argument `stackSize,` which is in bytes. The second argument, `entryPoint`, is a pointer to the function that is called every time this thread is to be executed. The function is analogous to the processes in SystemC. This function takes no arguments and returns no value. Just like in the SystemC CThread process, the function should contain an infinite loop with at least one call to function `wait()` or `wait_until()` to suspend the thread. It is necessary to suspend the thread within the infinite loop so that other threads get a chance to execute.

```
extern void wait()
```

This function does not take any arguments. When the function is called, control is returned to the scheduler and the thread is put into a suspended state. The thread will be rescheduled for execution in the next cycle. Execution will continue from the next line after the call to `wait().`

```
extern void wait_until( int (*wait_fn) () )
```

This function suspends the thread until a specified condition is true. It takes one argument, a pointer to a function. If the thread is suspended and waiting on a signal or condition, then this function is called at the beginning of every cycle to determine if the thread is to be scheduled or not. If the function passed as a parameter returns 1, then the thread is scheduled. If the function returns 0, then the thread is not scheduled.

This function can be used to wait on a signal. For example to wait on a signal `ready`, one has to write a function that reads the port connected to the signal *ready* and returns 1 if *ready* is asserted and 0 otherwise.

```
//code snippet to illustrate the use of wait_until()
//function to wait on a signal

Port ReadyIn;  //port to which the ready signal is
               //connected

//this function is called whenever a thread is waiting on
//the port ReadyIn.
int ready( )
{
     int val;
     portRead(&ReadyIn, &val);
     return val;
}

//inside the thread
threadA()
{
     …
     wait_until(&ready); //wait until signal ready is
asserted.
     …
}
```

6.4 GSM Model

The SystemC implementation of the GSM speech processing has already been discussed in the previous chapter. The handshake signals between modules and the module architecture were described. In the software implementation of the GSM model, the core processing functions were left untouched. The handshake signals, however, were optimized for speed. The interface between the modules is shown below in

Figure 13.

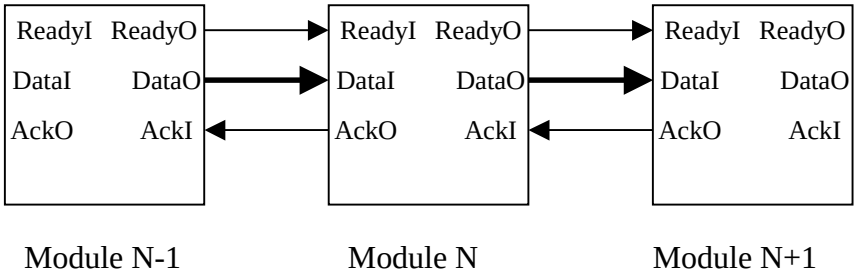


Figure 13 Handshake signals in software implementation.

When a module has data to send, it asserts the output signal *ReadyO* and writes the data on to the output bus *DataO*. It then waits on the signal *AckI*, which is an acknowledgement from the receiving module, before proceeding. Once it receives the acknowledgement, the sending module un-asserts the ready signal and waits until acknowledgement from the receiving module is un-asserted. A simple state diagram to illustrate the handshake is shown in the Figure 14.

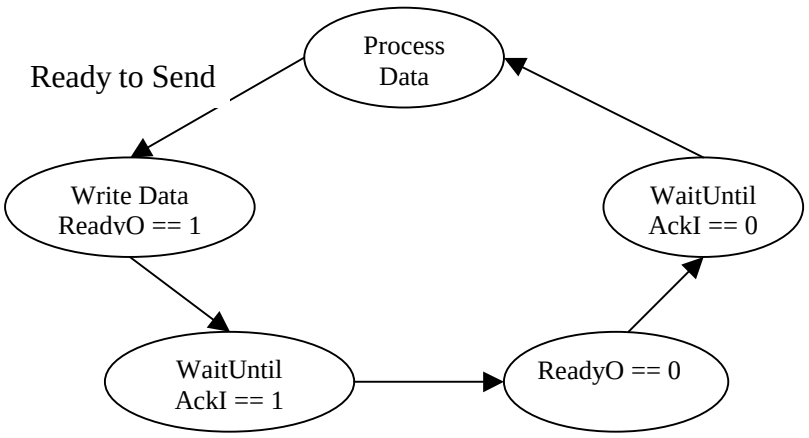


Figure 14 Handshake process on sending side.

On the receiving side, the receiver waits on the input signal *ReadyI*. Once this signal is asserted, the receiving module reads the data of the input bus *DataI*. After reading the data from the bus, it sends an acknowledgement back to sending module by asserting the signal *AckO*. It then waits until the sending module un-asserts its ready signal before proceeding to process the data. This handshake scheme ensures that sending module does not overwrite the data before a receiver has read the data and also ensures the receiver does not read same data twice. A simple state diagram to illustrate the handshake is shown in the Figure 15.

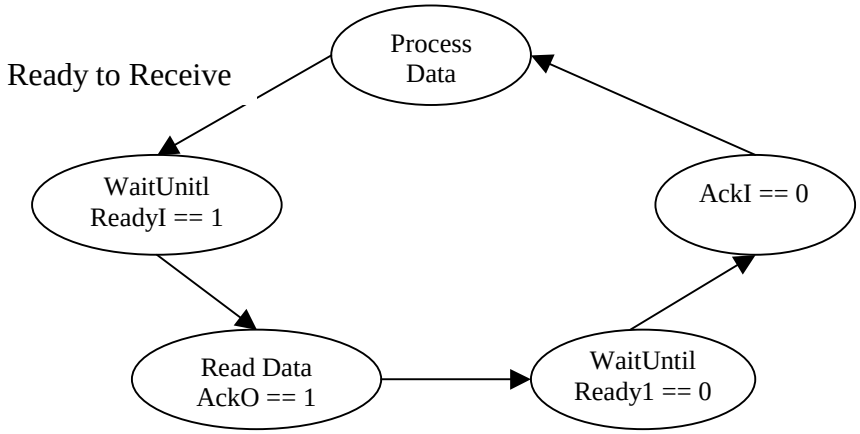


Figure 15 Handshake process on the receiving side.

The following code snippet further illustrates the use of signals to perform handshakes and synchronization. The code snippet is from the interleaving encoder module. This module receives data from the channel encoder module and sends data to the encryption module.

```
//a module shall declare only output signals..
Signal *interAck;
Signal *interReady;
Signal *interData;

Port interAckOut;
Port interReadyOut;
Port interReadyIn;
Port interAckIn;
Port interDataIn;
Port interDataOut;
..
..
//This is the function which is registered with the
//scheduler.
void interProcess()
{
```

```
    int i;

    //indicate that output data is not ready
    intPortWrite(&interReadyOut,0);

    wait();   //make sure other thread sees write

    //processing 4 segments only for example
    for(i=0;i<4;++i)
    {
        //wait until the data is ready
        wait_until(&waitOninterReadyIn);

        //acknowledge ready signal
        intPortWrite(&interAckOut,1);

        //Read data (bit array) from the input port
        portRead(&interDataIn, inter_input_data);

        //wait till the sender sees ACK = 1;
        wait_until(&waitOninterReadyIn0);

        //acknowledge data transfer
        intPortWrite(&interAckOut,0);

        //process data here
        inter_enc_process_data();

        //write out bit array
        portWrite(&interDataOut, interleaved_data);

        //indicate that output data is ready
        intPortWrite(&interReadyOut,1);

        // wait for Ack = 1 so we know other thread has
        //seen

        wait_until(&waitOninterAckIn);
        intPortWrite(&interReadyOut,0);

        // wait for Ack = 0 so we know that data has been
        //copied
        wait_until(&waitOninterAckIn0);
    }
}
```

In the code snippet, the function call wait_until() takes an argument which is a pointer to a function. The return value of the function determines if the thread will be moved from the suspended state to the active state by the scheduler. The code sample below illustrates the use of the *wait_until()* function call to wait on a signal *interAckIn*, with the use of function *waitOnIinterAckIn().*

```
// wait for Ack = 1
wait_until(&waitOnInterAckIn);
```

The function call *waitOnInterAckIn()* reads the port and returns the value of the signal connected to the port. The code snippet for the function is shown below.

```
//functions for the scheduler to wait on.
int waitOninterAckIn()
{
     int val;
     portRead(&interAckIn, &val);
     return val;
}
```

6.5 Modeling Guidelines

Before delving into the architecture and modeling guidelines, a brief description of some compatibility issues is presented below.

Features in SystemC that cannot be used in software implementations:

- Primitives: sc_method, sc_thread, processes sensitive to signals other than clock.
- Data Types: sc_int, sc_uint etc.

Features in the C language that cannot be used in synthesizable SystemC models:

- Pointers.
- Floating point data types.

Features in the C language that cannot be used in SystemC model for simulation:

- SystemC is a library of C++ classes to model hardware, and C++ is a superset of the C language. Hence, one should theoretically be able to use all the language constructs of C.

6.6 Suggested Organization

The current architecture of the modules is shown below in Figure 16. The input and output sections exist primarily to convert data from a word array to a bit array. The processing section contains all the functionality needed for processing the data. The processing section is primarily composed of C code and is encapsulated into a single C function call. Input data is copied into an input buffer and the function associated with

processing the data is called. Upon completing its task, the function writes the output into an output buffer. The module has various handshake and data signals going to other modules, which have been implemented using SystemC constructs.

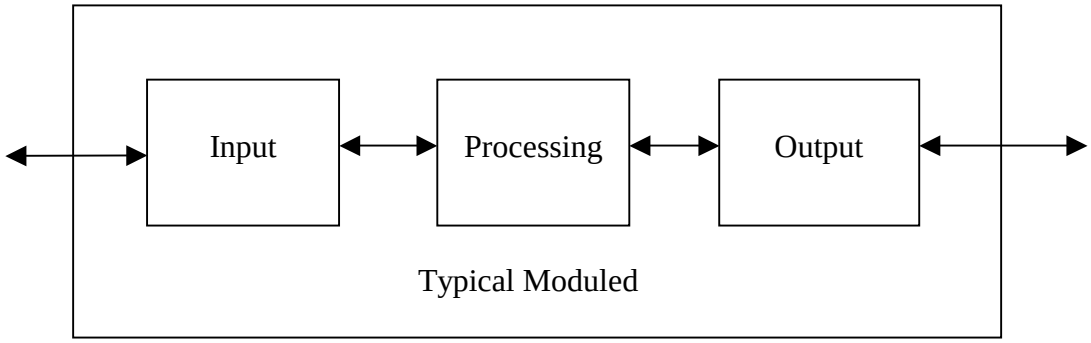


Figure 16 Current Architecture of the SystemC model of GSM speech processing.

The hardware synthesis tools and the software synthesis process are both compatible with restricted C code. The major part of the work in porting the GSM model to either hardware or software would involve porting various algorithms and data processing steps. Thus, if all the data processing within a module is encapsulated into a single C function or a few functions, then the design would lend itself well for both hardware and software synthesis. Another advantage of using native C code and data types for processing is that it decreases simulation time under SystemC. SystemC data types such as sc_int have overhead associated with them.

The control and communication aspects of the models should be implemented using SystemC signals and ports (Figure 17). But, all the modules should be restricted to using only Cthreads, as both the hardware tools and the software scheduler are not compatible with any of the other SystemC processes. Any changes made to the system will affect only the processing section that is implemented using the restricted C language (without pointers and float data types). This section is common to all the implementations of the model; i.e., abstract, hardware and software. Hence, any algorithmic or processing changes in the system get automatically updated in both the hardware and software implementation.

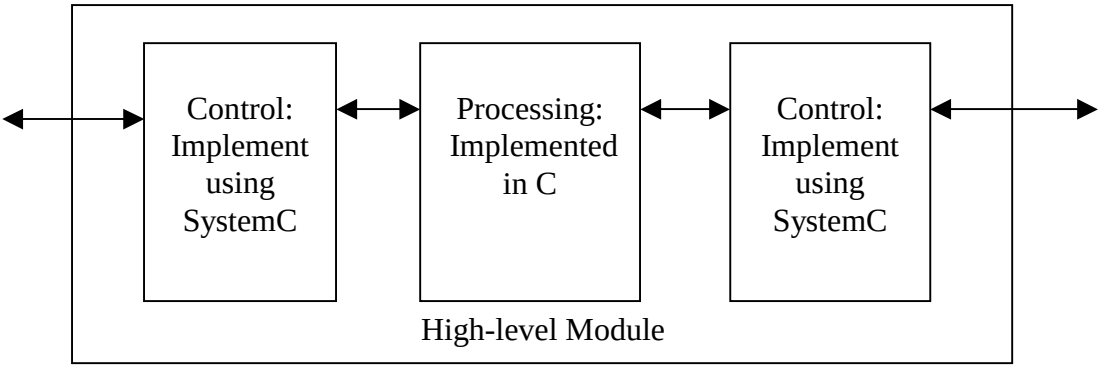


Figure 17 Hardware/software compatible module architecture..

Each of the modules should be coded in a separate file. All the variables and buffers used for data processing should not be declared as member variables of the SystemC class. In this way, the variable will be accessible to all the functions within a module and un-accessible from outside the module. Any communication or exchange of data between the modules should be restricted to using ports and signals. This will ensure that the software version of the model will function as intended.

The control section handles all the handshake signals and data transfers. This will make it easier to port the modules at the boundary of the hardware and software partition. At the boundaries, the software modules running on the embedded processor have to interact with the hardware modules implemented as ASIC logic. This will usually involve the implementation of specific driver software to interact with the hardware. Thus, having a processing section separate from the control section isolates the data processing algorithms from the hardware interface details.

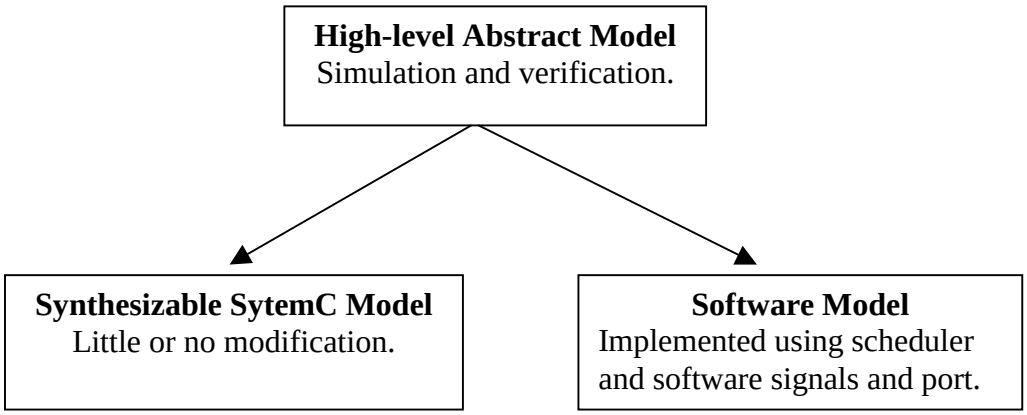


Figure 18 Model implementation flow.

In summary, a SystemC model following the above guidelines will be compatible with hardware synthesis tools and lend itself to embedded software implementation (Figure 18). Hardware synthesis requires little or no changes to the model. For software

synthesis, the modules have to be altered to make them compatible with the software. The amount of changes required is minimal and restricted to the handshake signals and control sections. Most of the alterations relate to changing from C++ syntax to C syntax and function calls.

7 Results

A software tool was developed by Pradeep Adhipathi [22] to partition a high-level model into hardware and software. Input to the tool is a representation of the model as a directed graph. The nodes of the graph represent the modules and the arcs represent signals between the modules. The tool also accepts timing restrictions and activation rates of the processes to arrive at the partition. The GSM speech-processing model was partitioned using the software tool. The resulting partition placed the speech encoder in hardware and the rest of the modules in software. The modules that were to be implemented in software were then ported to run as embedded software using the scheduler.

To evaluate and compare the performance of the embedded software implementation of GSM speech processing derived from the SystemC model, we need a reference implementation. GSM speech processing implemented in purely C from scratch is an ideal reference platform. Therefore, all the modules in the GSM speech processing that were to be implemented in software were ported to C manually. This implementation was a pure software implementation without a scheduler or signals. The model was executed for 4 speech segments and the timing was measured. Most of the modules processed information in bits. Hence, each bit had to be stored in a native C data type (ex. integer or character). To study the trade-offs of using character versus integer data types to store the bits, two models were implemented. The first model used the integer data type to represent each bit and the second model used the character data type to represent each bit. The time taken by each module to execute 4 speech segments is given in the table below.

Table 2 Execution times for the pure software implementation.

Processing time for 4 speech segments.			
Implementation	Machine Cycles	Instructions	Time (ms)
Pure software implementation using integers	1432757	947573	4.77
Pure software implementation using character.	1363869	892544	4.54

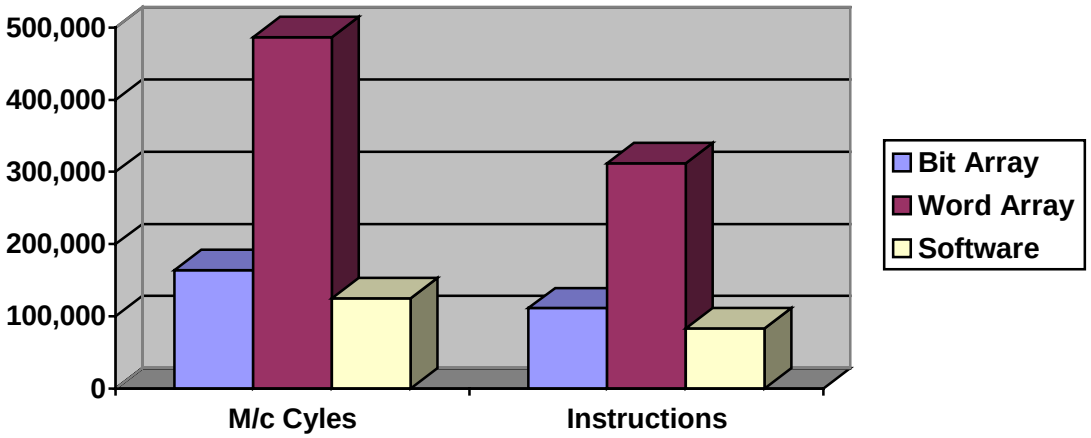
The software implemented with the character data type is faster than integer data type by 4.8%. Hence, the character data type implementation was used as the reference design and the SystemC derived models were evaluated against it.

In the SystemC model of the GSM speech processing, the modules processed the data in bit format (all modules except the speech encoder). The data was transferred from one module to another using a 16-bit bus. Thus, the data, which is a bit stream, was converted into a word array and transferred across to another module where it was converted back

into a bit stream before processing. In the software implementation the bits were stored using native data types such as integers or character, so one integer variable stores one bit. To transfer the data from one module to another in software, one could just transfer the bit array or convert the bit array into a word array and transfer the word array. This transfer would involve using the software signals and handshakes between the modules for reliable transmission. To compare the transfer methods, a part of the speech processing chain was implemented using both of the transfer methods. The parity encoder and the convolution encoder modules were implemented using both bit array transfers and word array transfers. The execution time for both implementations was measured and compared with the pure software implementation. The results for processing four speech segments are shown in the table below.

Table 3 Comparison of bit array and word array transfer models execution times.

Implementation	Number of Machine Cycles	Number of Instructions	Time (ms)
Bit array transfer	164,161	111,262	0.54
Word array transfer	486,606	312,072	1.62
Software Model	124,906	83,212	0.41



Overhead for WORD transfer model:
Computation load for 4 speech segments: 3.8 times software version
Overhead: 289% the computation for software version

Overhead for BIT ARRAY transfer model
Computation load for 4 speech segments: 1.31 times software version.
Overhead: 31% the computation for software version

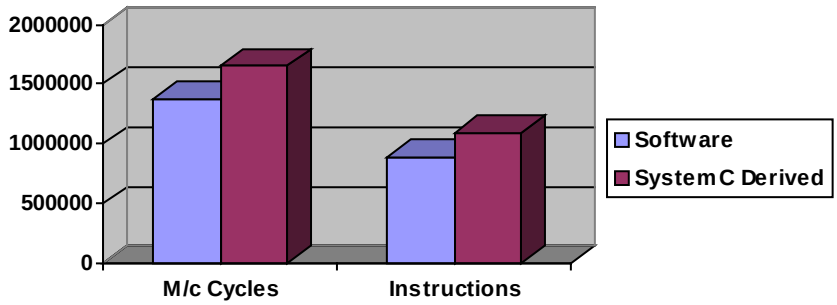
The large overhead in word transfer model was primarily due to the computational load of converting the bit array into a word array and vice versa. It is clear that any

implementation in software cannot use the word transfer model of the speech processing. Therefore, it was decided that the embedded software version would be implemented using bit arrays. The reference software model uses char arrays for storing bits and it was faster than the integer array model. Therefore, it was decided to store the bit array in a char data type for the SystemC derived software model to maintain consistency across the models.

The complete speech encoding chain, from parity encoding to differential encoding, was implemented in embedded software using the scheduler and software signals. The implementation was tested using the same speech samples that were used for testing the SystemC model. The output of every module was compared to the output of the corresponding SystemC module and was found to be identical. This proves that the software implementation using the scheduler and signals is accurate and identical to SystemC model. The time required to process four speech segments was measured and is tabulated in the table below.

Table 4 Comparison of pure software implementation and SystemC derived implementation

Processing time for 4 speech segments.			
Implementation	Machine Cycles	Instructions	Time (ms)
Pure software implementation using character.	1378784	892544	4.59
Software implementation derived from SystemC	1656004	1096601	5.52



.

The embedded software implementation with scheduler and software signals had a 20.1% overhead compared to the pure software implementation.

8 Conclusion

This thesis explored the idea of using SystemC to implement embedded software. A simple scheduler was proposed to implement SystemC models in software by scheduling and executing the SystemC clocked threads. Software constructs were developed to support signals. The scheduler and software signals were implemented and tested.

The SystemC model of the GSM speech processing was implemented as embedded software using the scheduler and software signals. The performance and overhead of this implementation was measured and compared with a pure software implementation of the system.

Initial results indicate that the idea of directly implementing embedded software from SystemC models is viable. The overhead of the scheduler would greatly reduce with more complex and computationally intensive modules.

This thesis looked into converting SystemC models to embedded C software. Future work can look into using C++ for embedded software implementation where a C++ compiler is available for the embedded processor. Emulation of hardware constructs like signals and ports is easier in C++ with its data encapsulation, function over-loading, and inheritance features. The models were manually ported to run on the embedded processor using the simplified scheduler. This process can be automated by developing tools for the synthesis of SystemC models into software using the scheduler.

Another direction that holds promise is the porting of a light SystemC kernel to the embedded processor. This is possible only if there is a suitable C++ compiler for the processor. The lightweight kernel should have all the syntax and semantics of the original kernel but without the overhead of the cycle accurate simulation requirements.

9 References

1. SystemC – A modeling platform supporting multiple design abstractions, Preti Rajan Panda, Synopsis Inc.
2. SystemC, www.systemc.org. (Current as of May 2002).
3. Cynlib: Forte Design Systems, http://www.forteds.com/products/cynlib.html (current as of March 2002).
4. Synopsys Inc, “Synopsys CoCentric SystemC Compiler”, http://www.synopsys.com/products/cocentric_systemC/cocentric_systemC.html (current as of March 2002).
5. J. R. Armstrong and Y. Ronen, Modeling with SystemC: A Case Study, 2000.
6. 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.
7. A. Varma, J. Armstrong, J. Baker, "A SystemC GSM Model for Hardware/Software Co-Design," International HDL Conference and Exhibition (HDLCon 2002), March 2002.
8. A. Varma, “Modeling and Synthesis with SystemC”, Master of Science thesis, Bradley Department of Electrical Engineering, Virginia Tech, 2001.
9. B. Sirpatil, J. Armstrong, J. Baker, "Using SystemC to Implement Embedded Software", International HDL Conference and Exhibition (HDLCon 2002), March 2002.
10. L. Green, “A5/1 Pedagogical Implementation”, http://jya.com/a51-pi.htm (current as of May 2002).
11. “Crack A5”, http://crypto.radiusnet.net/archive/cryptanalysis/crack-a5.htm (current as of May 2002)
12. B. Schneier, Applied Cryptography, Second Edition, John Wiley & Sons Inc, New York, 1996
13. “SystemC: Users Guide.”, Synopsys Inc, (www.systemc.org).
14. S.M. Redl, M.K.Weber, M.W.Oliphant, “An Introduction to GSM”, Artech House Inc, 1995.
15. StarCore SC140, http://www.starcore-dsp.com./
16. W. Mueller, J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, W. Rosenstiehl, "The Simulation Semantics of SystemC," Proceedings Design Automation and Test in Europe, 2001, pp. 64-70.
17. Raymond Steele, Mobile Radio Communications, IEEE Press, 1992.
18. T.S. Rappaport, Wireless Communications, Princeples and practices, Prentice Hall PTR, 1996.
19. Moore’s Law, http://www.intel.com/research/silicon/mooreslaw.htm.
20. Rochit Rajsuman, System-on-a-chip, Design and Test. Artech House, 2000.
21. T. Grotker, S.Liao, G.Martin, S.Swan, System Design with SystemC, Kluwer Academic Publishers, 2002.
22. J.R. Armstrong, P. Adhipathi, J.M. Baker, Jr., "Model and Synthesis Directed Task Assignment for Systems On a Chip," to be presented at the 15th International Conference on Parallel and Distributed Computing Systems (PDCS 2002), September 2002.

10 Appendix

```
//*****************************************************************************
// SystemC software scheduler -- This file contains the code for the
//  SystemC software scheduler.
//
// Author -- Mac Baker and Brijesh Sirpatil
//    Note -- much of this code was inspired by an 8101 RTOS scheduler
//          written by Craig Dry.
//*****************************************************************************

#ifndef SCHEDULER_H
#define SCHEDULER_H

// Integer signal and port
typedef struct int_signal_struct {
   int current_val;            /* current value of signal */
   int next_val;                        /* next value of signal */
} IntSignal;

typedef struct int_port_struct {
   IntSignal *signal;                   /* signal bound to this port */
} IntPort;

// Non-integer signal and port
typedef struct signal_struct {
   struct signal_struct *next;          // next signal in list
   int numBytes;                                // size of signal type
   int updateFlag;                              // whether signal has been updated or not
   void *current_val;                           // current value of signal
   void *next_val;                              // next value of signal
} Signal;

typedef struct port_struct {
   Signal *signal;                      // signal connected to port
} Port;


// function declarations
extern void SchedStart();
extern void SchedInit(int StackSize);
extern void createThread(int stackSize, void (*entryPoint)() );
extern void wait_until( int (*wait_fn) () );
extern void wait();

extern void ConnectPortToSignal(Port *port, Signal *signal);
extern Signal *CreateSignal(int numBytes);
extern void portRead(Port *port, void *dest);
extern void portWrite(Port *port, void *val);
extern int intPortRead(Port *port);
extern void intPortWrite(Port *port, int val);

//extern void ConnectPortToIntSignal(IntPort *port, IntSignal *signal);
//extern Signal *CreateIntSignal();
```

```
#endif


//*****************************************************************************
// SystemC software scheduler -- This file contains the code for the
//   SystemC software scheduler.
//
// Author -- Mac Baker and Brijesh Sirpatil
//    Note -- much of this code was inspired by (and taken from) an 8101
//          RTOS scheduler written by Craig Dry.
//*****************************************************************************

#include <stdio.h>                  // for error printing routine
#include <string.h>                 // for memcpy
#include "scheduler.h"

#define FALSE    0
#define TRUE     1
#define MAX_NUM_SIGNALS 256
#define MAX_NUM_THREADS 256

typedef struct thread_struct {
   int (*wait_fn) ();                       /* function for thread to wait on */
   int StatePtr;                /* pointer to thread's state */
   int StackPtr;                /* stack pointer for this thread */
   struct thread_struct *next;  /* pointer to next thread in list */
} Thread;


// External variables
// External functions
extern void __QCtxtSave();
extern void __QCtxtRestore();

// Global variables
static Signal *signals = NULL;              // signals in the system
static int numSignals = 0;                  // number of signals defined in system
static int numThreads = 0;                  // number of threads in the system
static Thread *activeList = NULL;           // list of active threads
static Thread *waitingList = NULL;                  // list of waiting threads
static Thread *currentThread = NULL;        // current active thread
static int NextStackStart;                  // next available address for a thread's stack

// Local function declarations
static void updateSignals();
static void updateSignal(Signal *sig);
static void activateWaitingThreads();
static Thread *addThread(Thread *threadList, Thread *this_thread);
static Thread *removeThread(Thread *threadList, Thread *this_thread);
static Thread *getNextThread(Thread *thisThread);
static int SuspendThread(Thread *thisThread);
static int SwitchToThread(Thread *thisThread);
static void error(char *str);
static void StartThread();
static void StopThread();
```

```
//---------------------------------------------------------------------
// CreateSignal -- Creates a new signal object.
// Input:
//    numBytes = size (number of bytes) of signal
//---------------------------------------------------------------------
Signal *CreateSignal(int numBytes)
{
  Signal *sig;

  // allocate the signal struct
  sig = (Signal *) NextStackStart;
  NextStackStart += sizeof(Signal);

  // allocate the current and next values of signal
  sig->current_val = (void *) NextStackStart;
  NextStackStart += numBytes;

  sig->next_val = (void *) NextStackStart;
  NextStackStart += numBytes;

  // re-align the stack
  NextStackStart = (NextStackStart + 8) & ~0x7;

  // initialize rest of signal structure
  sig->updateFlag = FALSE;
  sig->numBytes = numBytes;

  // add signal to list
  sig->next = signals;
  signals = sig;

  // increment number of signals
  numSignals++;

  return sig;
}



//---------------------------------------------------------------------
// ConnectPortToSignal -- Logically connects (binds) a port to a signal.
//
// Input:
//          port -- port data structure
//          signal -- signal data structure
//---------------------------------------------------------------------
void ConnectPortToSignal(Port *port, Signal *signal)
{
   // connect port to signal
   if ((port != NULL) && (signal != NULL)) {
      port->signal = signal;
   }
}
```

```
//---------------------------------------------------------------------
// updateSignal -- Sets a signal's current value equal to its next value.
// Input:
//    sig = pointer to signal
//---------------------------------------------------------------------
void updateSignal(Signal *sig)
{
  // copy numBytes from next_val to current_val
  memcpy( sig->current_val, sig->next_val, sig->numBytes);
}


//---------------------------------------------------------------------
// portRead -- Reads the current value of the signal connected to a port,
//  storing the value in a given destination.
// Input:
//   port = pointer to port
// Output:
//   dest = pointer to memory location where value will be stored
//---------------------------------------------------------------------
void portRead(Port *port, void *dest)
{
   // copy numBytes from current_val to dest
   memcpy( dest, port->signal->current_val, port->signal->numBytes);
}


//---------------------------------------------------------------------
// portWrite -- Sets the next value of the signal connected to the port
//   to the given value.
// Input:
//    port = pointer to port
//    val = pointer to value to be written
//---------------------------------------------------------------------
void portWrite(Port *port, void *val)
{
  // copy numBytes from val to next_val
  memcpy( port->signal->next_val, val, port->signal->numBytes);
  port->signal->updateFlag = TRUE;
}

//---------------------------------------------------------------------
// intPortRead -- Reads the current value of the signal connected to a port,
//  storing the value in a given destination.
// Input:
//   port = pointer to port
// Output:
//   returns current value of integer signal
//---------------------------------------------------------------------
int intPortRead(Port *port)
{
  return  *((int *)(port->signal->current_val));
}

//---------------------------------------------------------------------
// IntPortWrite -- Sets the next value of the signal connected to the port
```

```
//   to the given value.
// Input:
//    port = pointer to port
//    val = integer value to be written
//---------------------------------------------------------------------------
void intPortWrite(Port *port, int val)
{
  *((int *)(port->signal->next_val)) = val;
  port->signal->updateFlag = TRUE;
}



//---------------------------------------------------------------------------
// wait -- Suspends the current thread and switches to the next active thread.
// If all active threads have been processed for this cycle, then the signals
// will be updated before re-executing the first active thread.
// Input -- none
//---------------------------------------------------------------------------
void wait()
{
   // suspend the current thread
   if (!SuspendThread(currentThread)) {

      // get the next ready thread (updating signals if necessary)
      currentThread = getNextThread(currentThread);

      // switch to the new thread
      SwitchToThread(currentThread);
   }
}



//---------------------------------------------------------------------------
// wait_until -- Suspends the current thread and switches to the next active
// thread. Similar to wait(), the signal will be updated if this was the last
// thread to execute during this cycle. Also, if the condition function evaluates
// to FALSE, the suspending thread will be placed in the WAITING state; otherwise,
// it will remain in the ACTIVE state.
// Input -- condition_fn = function specifying condition thread waits for
//---------------------------------------------------------------------------
void wait_until( int (*wait_fn) () )
{
   Thread *thisThread;

   // attach wait_fn to this thread
   currentThread->wait_fn = wait_fn;

   // remember this thread
   thisThread = currentThread;

   // get the next thread to execute
   currentThread = getNextThread(thisThread);

   // suspend current thread
```

```
    if (!SuspendThread(thisThread)) {

      // if condition is false, change thread to WAITING state
      if ( !wait_fn() ) {
         // remove thread from ActiveList
         activeList = removeThread(activeList, thisThread);

         // add thread to WaitingList
         waitingList = addThread(waitingList, thisThread);

         // if this was the only thread, then try to find another one
         if (currentThread == thisThread) {
           currentThread = getNextThread(activeList);
         }
      }

      // switch to next thread
      SwitchToThread(currentThread);
    }
}


//---------------------------------------------------------------------
// updateSignals -- Updates the values of all signals in the system.
//---------------------------------------------------------------------
void updateSignals()
{
  Signal *sig;

  for (sig = signals; sig != NULL; sig = sig->next) {
    if (sig->updateFlag == TRUE) {
      updateSignal(sig);
      sig->updateFlag = FALSE;
    }
  }
}



//---------------------------------------------------------------------
// activateWaitingThreads -- Tests the condition function for each waiting
//  thread. If the condition is true, then the thread is removed from the
// WaitingList and added to the ActiveList of threads.
//---------------------------------------------------------------------
void activateWaitingThreads()
{
   Thread *this_thread, *next_thread;

   // for each thread, test if thread is now ready
   this_thread = waitingList;

   while (this_thread != NULL) {
      if ( this_thread->wait_fn() ) {
         // thread is now ready, so remember next thread in list
         next_thread = this_thread->next;
```

```
            // remove from WaitingList, add it to ActiveList
            waitingList = removeThread(waitingList, this_thread);
            activeList = addThread(activeList, this_thread);

            // repeat for next_thread
            this_thread = next_thread;
        }
        else {
                        // thread still not ready, so check next thread
            this_thread = this_thread->next;
        }
    }
}


//---------------------------------------------------------------------
// addThread -- Adds a thread to a list of threads. Note that the threads
// are not ordered, so we can simply add thread to beginning of the list.
// Input:
//    threadList -- pointer to list of threads
//    this_thread -- thread to be added to list
// Output:
//    threadList -- pointer to updated list of threads
//---------------------------------------------------------------------
Thread *addThread(Thread *threadList, Thread *this_thread)
{
    // add thread to beginning of list
    this_thread->next = threadList;

    // return pointer to thread (new head of list)
    return this_thread;
}


//---------------------------------------------------------------------
// removeThread -- Searches through a list of threads until it finds
// the given thread. It will then remove the thread from the list.
// Input:
//     threadList -- pointer to list of threads
//     this_thread -- thread to be removed
// Output:
//     threadList -- pointer to updated list of threads
//---------------------------------------------------------------------
Thread *removeThread(Thread *threadList, Thread *this_thread)
{
    Thread *thread_ptr;

    // first, see if thread is the first one in the list
    if (this_thread == threadList) {
        // remove thread from list, return rest of list
        thread_ptr = this_thread->next;
        this_thread->next = NULL;
        return thread_ptr;
    }

    // else, search through the list until we find this_thread
```

```
   thread_ptr = threadList;
   while ((thread_ptr != NULL) && (thread_ptr->next != this_thread)) {
      thread_ptr = thread_ptr->next;
   }

   // did we find it?
   if (thread_ptr != NULL) {
      // must have, so remove thread from list
      thread_ptr->next = this_thread->next;
      this_thread->next = NULL;
   }

   // return pointer to updated list
   return threadList;
}



//---------------------------------------------------------------------------
// getNextThread -- Returns the next active thread. If all the threads have
// been processed for this cycle, then first the signals are updated, and then
// any waiting threads that are now ready are activated.
// Input:
//    thisThread -- pointer to current active thread
//          activeThreads -- list of active threads
//          waitingThreads -- list of waiting threads
// Output:
//          nextThread -- next thread to execute
//---------------------------------------------------------------------------
Thread *getNextThread(Thread *thisThread)
{
   Thread *nextThread;

   if (thisThread != NULL) {
      nextThread = thisThread->next;
   }
   else {
      nextThread = NULL;
   }

   if (nextThread == NULL) {
      // we've executed everything this cycle, so we need to update
            //          signals and waiting threads
      updateSignals();
      activateWaitingThreads();

      // next thread will be first thread in active list (if there are no
      //    active threads, we'll keep updating signals and activate waiting
      //    threads until there is one)
      nextThread = activeList;

      if (nextThread == NULL) {
         printf("No more active threads.\n");
         exit(1);
      }
   }
```

```
   // if we got here, there must be an active thread
   return nextThread;
}


//---------------------------------------------------------------------------
// SuspendThread -- Saves the environment of the current thread. This gets
// a little hairy because we have to mix some C and assembly, and also keep
// track of what's on the stack. If this routine is ever modified, you may
// need to update the assembly-language part of it because that code expects
// a certain-sized stack frame. If you modify the C-code, you may be changing
// the size of the stack frame. Look at the assembly .sl file to see.
// Input:
//     currentThread -- pointer to current thread
// Output:
//           Returns 0 upon invocation, returns non-zero when thread is resumed
//                       (see SwitchToThread for more details)
//---------------------------------------------------------------------------
int SuspendThread(Thread *thisThread)
{
   static int currSP;                       // current stack pointer
   static int currStatePtr;    // current state pointer

   // disable interrupts while suspending thread
   asm("di");

   // save this thread's SP
   asm("tfra sp,r4");                              // save SP in r0
   asm("move.l r4,__currSP");        // save r0 in currSP
   thisThread->StackPtr = currSP;

   // save the context of this thread
   currStatePtr = thisThread->StatePtr;         // pointer to state info
   asm("move.l __currStatePtr,r4");             // load pointer into SP
   asm("tfra r4,sp");
   __QCtxtSave();

   // restore the SP
   asm("move.l __currSP,r4");
   asm("tfra r4,sp");

   // duplicate the stack frame -- if you change the C-code,
   //  this will probably need to be updated.
   asm("adda #24,sp");
   asm("move.l (sp-56),d4");          // wait_fn
   asm("move.l d4,(sp-24)");
   asm("move.l (sp-52),d4");          // thisThread
   asm("move.l d4,(sp-20)");
   asm("move.l (sp-48),d4");          // return value
   asm("move.l d4,(sp-16)");
   asm("move.l (sp-40),d4");          // return address
   asm("move.l d4,(sp-8)");
   asm("move.l (sp-36),d4");          // return flag reg
   asm("move.l d4,(sp-4)");
```

```
    // re-enable interrupts
    asm("ei");

    // return 0 (when this thread is restored, it will return 1 -- see SwitchToThread
    //   for details)
    asm("clr d0");
    asm("rtstk");

    // this is just to keep the compiler happy. It never executes
    return 0;
}



//---------------------------------------------------------------------------
// SwitchToThread -- Restores the environment of a suspended thread. The
// thread will resume execution as if it has returned from a call to
// SuspendThread() with a return value of 1.
// Input:
//           currentThread -- pointer to thread to switch to
//---------------------------------------------------------------------------
int SwitchToThread(Thread *thisThread)
{
    static int SP;                  // SP of currentThread
    static int StatePtr; // state pointer of currentThread

    // disable interrupts while we're restoring thread
    asm("di");

    // remember threads SP
    SP = thisThread->StackPtr;

    // restore state ptr for this thread
    StatePtr = thisThread->StatePtr + 34*2*sizeof(int);
    asm("move.l <__StatePtr,r4");
    asm("tfra r4,sp");

    // restore the thread's context
    __QCtxtRestore();

    // restore stack ptr for this thread
    asm("move.l __SP,r4");
    asm("tfra r4,sp");

    asm("suba #8,sp");                      // clean up local vars off stack

    // re-enable interrupts
    asm("ei");

    // set return value to 1
    asm("move.l #1,d0");

    // return
    asm("rtstk");

    // this is here to keep the compiler happy. It never gets executed
```

```
   return 1;
}



//---------------------------------------------------------------------------
// createThread -- Creates a new thread, adding the thread to the list of
// active threads.
// Input:
//           stackSize -- size of the stack needed for this thread
//           entryPoint -- function to be executed by thread
//---------------------------------------------------------------------------
void createThread(int stackSize, void (*entryPoint)() )
{
  Thread *newThread;
  int *SP;

  // allocate the signal struct
  newThread = (Thread *) NextStackStart;
  NextStackStart += sizeof(Thread);
  NextStackStart = (NextStackStart + 8) & ~0x7;

  newThread->wait_fn = NULL;
  newThread->next = NULL;

  // increment number of threads
  numThreads++;

  // add thread to active list
  activeList = addThread(activeList, newThread);

  // allocate space for the thread's state
  newThread->StatePtr = NextStackStart;
  NextStackStart += 35*2*sizeof(int);

  // allocate a stack for the thread
  SP = (int *) NextStackStart;
  NextStackStart += stackSize;
  NextStackStart = (NextStackStart + 8) & ~0x7;       // stack must be aligned
                                                                  //   on 8-byte boundary

  // set up stack to execute thread. The thread will initially execute
  // the StartThread function, then the entryPoint() function, and (if
  // entryPoint() ever returns), the StopThread function.
  *(SP++) = (int) &StopThread;      // return address
  *(SP++) = 0x000c000c;                       // status register:
                                                                  //   disable interrupts, use
                                                                  //     ESP,  2's  complement
rounding,
                                                                  //   saturation mode
  *(SP++) = (int) entryPoint;                             // return address
  *(SP++) = 0x0004000c;                       // status (same as before, but
                                                                  //   enable interrupts)
  *(SP++) = (int) &StartThread;     // the first return will jump here
  *(SP++) = 0x000c000c;
```

```
  SP++;                                          // dummy stack vars
  SP++;

  newThread->StackPtr = (int) SP;

}


//---------------------------------------------------------------------
// SchedInit -- This function initializes the data structures used by the
// scheduler. It figures out a "relatively safe" place in memory to put
// the threads' stacks and initializes the lists of active threads and
// waiting threads.
// Input:
//          StackSize -- amount of stack space (in bytes) to reserve for the
//                                  main program. The threads' stacks will be located
//                                  after this in memory
//---------------------------------------------------------------------
void SchedInit(int StackSize)
{
   // initially, set NextStackStart to the current stack pointer value
   asm("tfra sp,r4");                                  // r0 = current SP
   asm("move.l r4,_NextStackStart");          // save in NextStackStart

   // now, make sure we reserve at least StackSize bytes for the main program
   NextStackStart += StackSize;

   // stack must be aligned on 8-byte boundary
   NextStackStart = (NextStackStart + 8) & ~0x7;
}


//---------------------------------------------------------------------
// SchedStart -- This routine starts the scheduler. It is assumed that
// SchedInit has already been called, and also that the threads have been
// created. This routine will transfer control to one of the active threads.
// Input: none
//---------------------------------------------------------------------
void SchedStart()
{
  // make head of active list the current thread
  currentThread = activeList;

  // now switch to current thread
  SwitchToThread(currentThread);
}



//---------------------------------------------------------------------
// error -- Prints an error message on stderr and exits.
// Input:
//    str = string to be printed
//---------------------------------------------------------------------
void error(char *str)
{
```

```
  fprintf(stderr, "ERROR: %s\n", str);
  exit(1);
}


//--------------------------------------------------------------
// StartThread -- does nothing
//--------------------------------------------------------------
void StartThread()
{
  asm("rte");
}


//--------------------------------------------------------------
// StopThread -- does nothing
//--------------------------------------------------------------
void StopThread()
{
  Thread *nextThread;

  // get next thread
  nextThread = getNextThread(currentThread);

  // kill this thread
  activeList = removeThread(activeList, currentThread);

  // switch to next thread
  if (activeList != NULL) {
    currentThread = nextThread;
    SwitchToThread(currentThread);
  }

  // if we get here, there are no more active threads, so exit
  exit(0);
}


//****************************************************************************
// SystemC software scheduler -- This file contains the code to test the
//   SystemC software scheduler.
//
// Author -- Mac Baker and Brijesh Sirpatil
//****************************************************************************

#include <stdio.h>                  // for printf
#include "scheduler.h"

Signal *Ready;                      // handshake signals
Signal *Ack;

Port InA;                                   // input and output ports for ThreadA
Port InB;                                   //   and ThreadB
Port OutA;
Port OutB;
```

```
int count = 0;                          // counter passed between threads


//
// Wait functions -- to use wait_until(cond), you have to have
//                   a function that tests the condition. It should return
//                   1 if the condition is true, 0 if false.
//
int Ack0() {                    // wait until Ack == 0
  int val;

  portRead(&InA, &val);
  return !val;
}

int Ack1() {                    // wait until Ack == 1
  int val;

  portRead(&InA, &val);
  return val;
}

int Ready0() {                  // wait until Ready == 0
  int val;

  portRead(&InB, &val);
  return !val;
}

int Ready1() {                  // wait until Ready == 1
  int val;

  portRead(&InB, &val);
  return val;
}


//---------------------------------------------------------------------
// Thread A code -- This thread updates the shared counter,
//          and then initiates the handshake (notifies other thread
//          that count is ready and waits for acknowledgement).
//---------------------------------------------------------------------
void ThreadA()
{
  int temp;
  int i;

  // set Ready to 0
  temp = 0;
  portWrite(&OutA, &temp);

  wait();             // make sure other thread sees write

  for (i=0; i < 10; i++) {
    // increment the counter (i.e., update shared data)
    count++;
```

```
    // set Ready to 1
    temp = 1;
    portWrite(&OutA, &temp);

    // wait for Ack = 1 so we know other thread has seen
    //      Ready
    wait_until( &Ack1 );

    // set Ready to 0
    temp = 0;
    portWrite(&OutA, &temp);

    // wait for Ack = 0
    wait_until( &Ack0 );
  }

  exit(0);

}


//---------------------------------------------------------------------
// Thread B code
//---------------------------------------------------------------------
void ThreadB()
{
  int temp;
  int i;

  // set Ack = 0
  temp = 0;
  portWrite(&OutB, &temp);

  for (i = 0; i < 10; i++) {
    // wait for Ready = 1
    wait_until( &Ready1 );

    // set Ack = 1 to acknowledge that we saw data
    temp = 1;
    portWrite(&OutB, &temp);

    // read updated count value
    printf("Thread B got Ready signal %d.\n", count);

    // wait for Ready = 0
    wait_until( &Ready0 );

    // set Ack = 0
    temp = 0;
    portWrite(&OutB, &temp);
  }

  exit(0);

}
```

```
//
// Main code
//
void main()
{
  SchedInit(1024);

  // create the signals
  Ready = CreateSignal(sizeof(int));
  Ack = CreateSignal(sizeof(int));

  // connect the signals to the ports
  ConnectPortToSignal(&OutA, Ready);
  ConnectPortToSignal(&InA, Ack);

  ConnectPortToSignal(&OutB, Ack);
  ConnectPortToSignal(&InB, Ready);

  // create the threads
  createThread(1024, &ThreadA);
  createThread(1024, &ThreadB);

  // start the scheduler
  SchedStart();
}
```

Vita

Brijesh Sirpatil was born on April 5, 1976 in Gulbarga, a city in the state of Karnataka in India. He graduated with a Bachelor of Engineering degree in plied Electronics and Communication Engineering from Gulbarga University in the year 1998. After working in the Robotics and Automation industry for more than a year, he decided to pursue his higher studies in the field of computer Engineering. He graduated with a Master of Science degree in Electrical Engineering from Virginia Tech in Summer of 2002.