8-2004

Development of SystemC Modules from HDL for System-on-Chip Applications

Siddhartha Devalapalli
University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a thesis written by Siddhartha Devalapalli entitled "Development of SystemC Modules from HDL for System-on-Chip Applications." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Dr. Donald W. Bouldin, Major Professor

We have read this thesis and recommend its acceptance:

Dr. Gregory D. Peterson, Dr. Chandra Tan

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a thesis written by Siddhartha Devalapalli entitled "Development of SystemC Modules from HDL for System-on-Chip Applications". I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Dr. Donald W. Bouldin

Major Professor

We have read this thesis and
recommend its acceptance:

Dr. Gregory D. Peterson

Dr. Chandra Tan

Accepted for the Council:

Anne Mayhew

Vice Chancellor and
Dean of Graduate Studies

(Original signatures are on file with official student records.)

Development of SystemC Modules from HDL for System-on-Chip Applications

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Siddhartha Devalapalli

August 2004

Dedicated to my school

The Hyderabad Public School, Hyderabad. India.

Acknowledgements

Foremost, I would like to express my deepest gratitude to my advisor, Professor Dr. Donald Wayne Bouldin, for his excellent guidance and endless support during my graduate study and research at The University of Tennessee, Knoxville.

My many thanks to Professor Dr. Gregory Peterson and Dr. Chandra Tan for serving on my thesis committee.

Special thanks to Dr. Chandra Tan for the help extended to me whenever I approached him with a problem.

I also thank Dr. Wilson Synder, the developer of *Verilator* for his replies to my e-mail queries.

I'm deeply indebted to my parents for the support and the encouragement they provided to me to explore higher levels of education.

I am especially grateful to the Department of Electrical and Computer Engineering for providing me with a teaching assistantship during the course of my study, not to mention of the wonderful office with cable TV allotted to me. A million thanks will fall far short of thanking Professor Emeritus Dr. Walter Green enough for his support during and for my assistantship.

Many thanks to my friends Teja, Ashwin, Sampath and Saumil for all their help.

Abstract

Designs are becoming bigger in size, faster in speed and larger in complexity with the emergence of System-on-Chip designs. Hardware components and embedded software co-exist in such designs making it necessary to describe designs at higher levels of abstraction. Describing designs at higher levels of abstraction enables faster simulation, hardware software co-simulation and architectural exploration. SystemC is a solution.

Also the time to market can be reduced greatly if previously designed intellectually property (IP) described in a hardware description language (HDL) can be reused in SystemC.

In this research SystemC modules were successfully developed and verified from Verilog using existing tools. *Verilator* is a tool that translates synthesizable Verilog into C++ or SystemC code with certain limitations. The generated SystemC code was simulated and compared against the simulation results of the original Verilog code. To add credibility several simple and a few large cores were transformed into SystemC and the simulation results of the Verilog and the SystemC code were compared.

Pure SystemC code was synthesized using Synopsys' SystemC Compiler and DC Shell and an area report based on the TSMC 0.18 technology was generated. The synthesis produced a Verilog code, which was simulated, and the simulations

compared to the SystemC simulations

Available VHDL to Verilog and vice-versa conversion tools were also investigated. Mentor Graphics' HDL Designer and Ocean Logic's VHDL to Verilog converter were used for this purpose. Simulation results before and after the conversions were compared.

Table of contents

List of figures

1. Introduction

The current mainstream technology employed to capture the design of a complex integrated circuit is based on the use of hardware description languages like Verilog and VHDL. Circuits can be described in abstractions varying from gate level to algorithmic behaviors using these languages. HDL design starts with the specification from which a detailed clock cycle accurate hardware description has to be developed; but this is not always possible especially when the size of the design contains millions of gates and embedded software.

1.1. System-on-Chip (SoC)

Conventionally all designs involved the development of a medium complexity integrated circuit of about 500,000 gates [30]. They were essentially made up of core logic and some hard macros, like on-chip memory. With rapid advances in semiconductor processing technology the density of gates on the die increased in line with what Moore's law predicted. This helped the realization of more complicated designs on the same IC.

Over the last few years, with the advent of bleeding edge technology like the third generation (3G) mobile communication services offering Internet through mobile phones, an increasing need has been that of incorporating the traditional microprocessors, memories and peripherals, all on a single chip, see Figure 1. This is

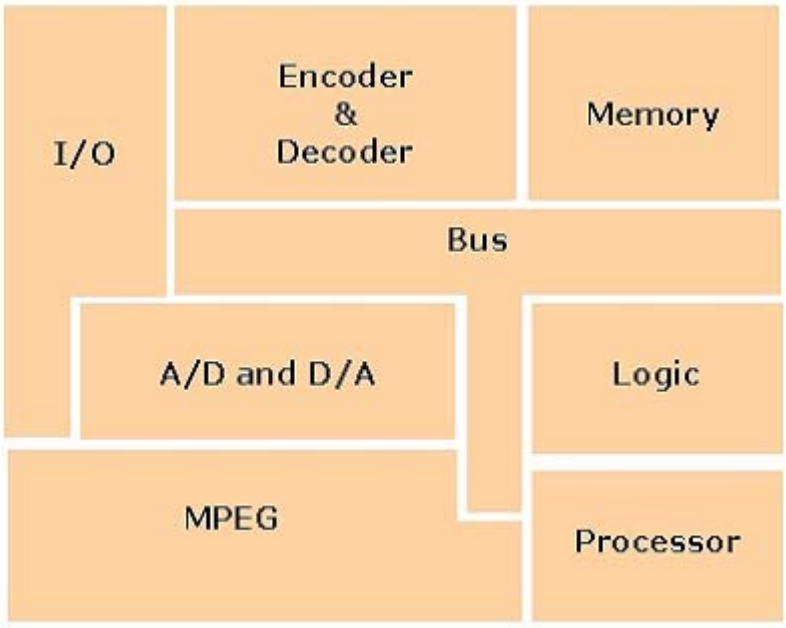

Figure 1 Typical SoC design

what has been marked as the beginning of the SoC era. A SoC usually contains one or more programmable processors, on-chip memory, timers, interrupt controllers, busses, specifically designed complex hardware modules and embedded software.

Essential resources of every SoC are intellectual property cores. An IP core is a complex module that performs a specific task and is created for reuse. These IP cores are the building blocks for SoCs and occupy a very high percentage of the area of the SoC.

1.2. System-level design

A system-level design language contains the semantics for describing a system prior to mapping it onto an architecture. Components must be able to be

described without making assumptions about the implementation [23]. Take, for example, a microprocessor-based system. The system-level design language must be able to describe the behavior of the components in a system irrespective of whether they will be mapped to software running on a microprocessor, or to application-specific hardware. A performance model of the system must be constructible in the language enabling the exploration of the architecture without specifying the exact microprocessor or the bus architecture that will be used to implement the system. Based on the results provided by this model the best processor and the best bus architecture can be selected.

In order to bridge the gap between the growing complexity of chip design (see Figure 2) and increased time-to-market pressures it is required that the design process be styled to higher levels of abstraction and the re-use of pre-designed complex system components.

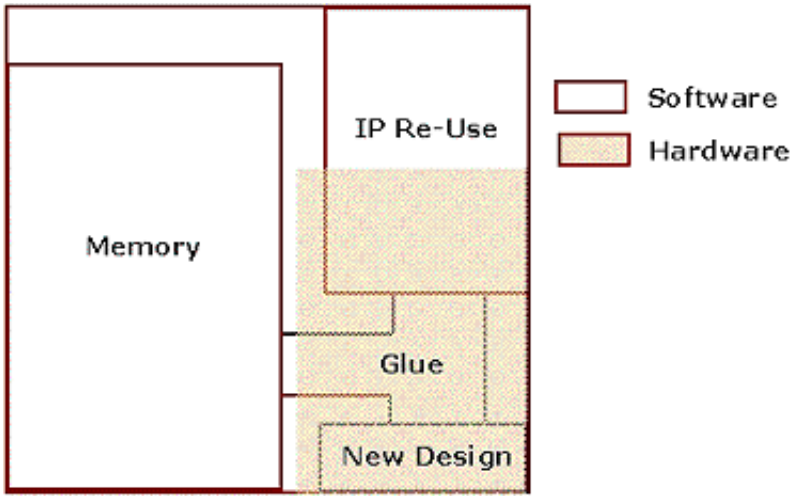

Figure 2 System content is being dominated by software

Also as software has become a much more important part of all VLSI design systems, a system-level design language is needed to handle both hardware and software, see Figure 2. Essentially an SLDL supports modeling at all levels of abstraction and these models should be simulatable in order to verify the functionality and timing constraints of a model.

1.3. SystemC

With systems and software engineers programming in C or C++ and their hardware colleagues working in hardware description languages like Verilog and VHDL, problems arising from the use of different design languages, incompatible tools and fragmented tool flows are becoming common.

SystemC 2.0 is a modeling language based on C++. It extends the capabilities of C++ by allowing for the modeling of hardware descriptions. It adds for the modeling of hardware descriptions, it adds a class library of functions, data types and other language constructs to C++. This class library contains powerful new mechanisms that allow design teams to model and verify designs expressed at true system levels of abstraction, refine these to reflect implementation choices and finally link the system model to hardware design implementations and verification.

SystemC is a single language to define, co-simulate, refine a system of hardware and software components down to the register transfer level for synthesis. SystemC provides a kernel for the simulation of a system executable specification

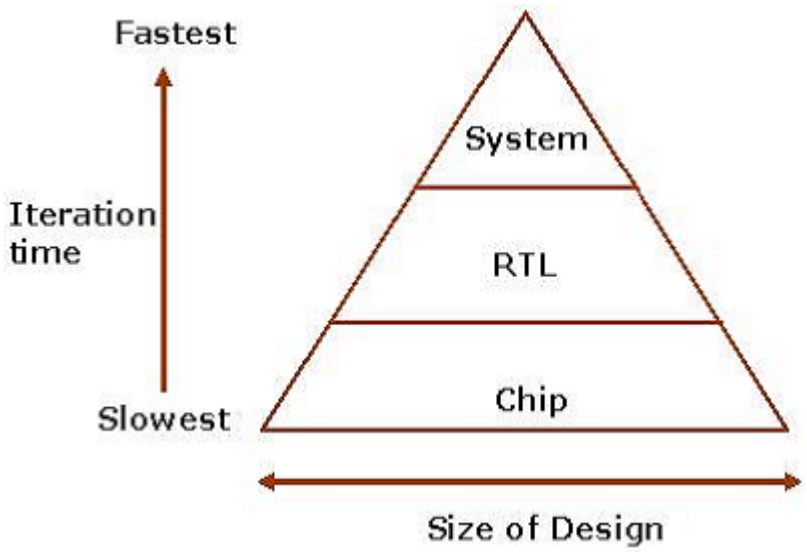

Figure 3 Exploring designs at different levels of abstraction (J. Bhasker)

written in SystemC [28]. It can be used to describe cycle accurate models of system-level design software algorithms and hardware architecture.

1.3.1. Why SystemC

When a system is expressed at the system level, it is easier to iterate and explore various algorithms as compared to exploring at the register level or gate level, Figure 3. The design at the chip level is relatively very large when compared to the design at the system level. The whole system can be described using SystemC and this becomes the executable model for the hardware design team, which iteratively refines the system model down to the register transfer level to enable synthesis. The hardware design process becomes a refinement of the specification.

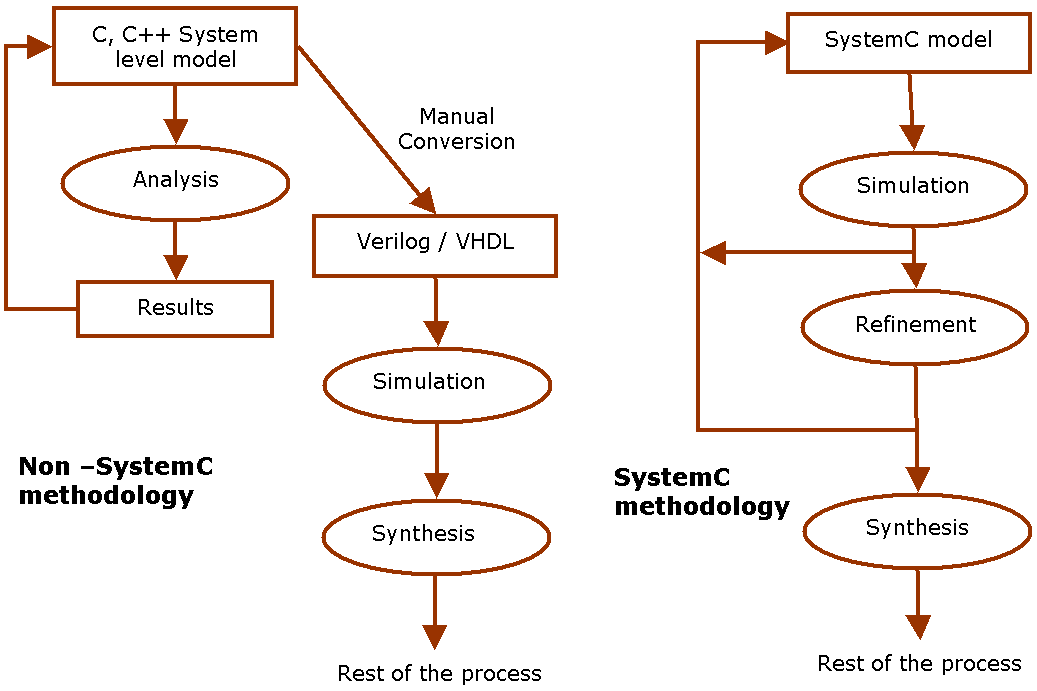

Figure 4 SystemC and non-SystemC methodology

1.3.2. SystemC design methodology

To understand the SystemC design methodology and its advantages it is important to understand the non-SystemC methodology.

In the non-SystemC methodology, Figure 4, the system designers would write the executable specifications in C or C++ and then verify and debug this design. After finding that this design satisfied all specifications it is handed over to the RTL design group. The RTL design group then re-writes the design at the RTL level to synthesize it to gates. In such a methodology the functionality of the RTL description sometimes varies from the executable specifications and consequently becomes prone to error. Also, it is a real problem if at the RTL level it is discovered

that something in the conceptual model cannot be implemented, as there is no common design environment between the system design and design implementation.

In the SystemC methodology, Figure 4, a system designer need only write a SystemC model. The designer can iteratively refine the executable specifications down to the register transfer level, which is still in SystemC, prior to synthesis. The testbench can be reused to ensure that the iterative process has not produced any errors. If during the RTL implementation, it is discovered that something is conceptually wrong, it is much easier to rewrite it.

1.4. Scope of this thesis

The work presented in this thesis is the initial work of a larger project in which a complete system-on-chip will be implemented using SystemC. Several translation tools were investigated by running tests on them. The best among them that were available for free were selected. This thesis also tried to make available a setup here at the University of Tennessee where VHDL code could be converted to SystemC that can be used for simulation. A tutorial to that extent has been prepared.

Existing IP in HDL was converted to SystemC using existing tools. *Verilator* is an existing tool that converts synthesizable Verilog code to SystemC. VHD2VL is a tool developed by Ocean Logic Limited that converts a large subset of the synthesizable VHDL code into Verilog. Large cores like the CORDIC algorithm and small cores like flip-flops, multiplexers, demultiplexers and adders were converted to

SystemC and the simulation results were compared with the simulation results of the Verilog code.

2. Background

The idea of converting available HDL IP cores into SystemC / C++ has been actively pursued in academic research. Professor Philip Wisley of the University of Cincinnati developed the SAVANT tool that converts VHDL to C++. Using SAVANT and by adding features to it Professor Franco Fummi at the University of Verona, Italy developed a tool that would output a SystemC code. Both these tools followed the Internal Intermediate Representation (IIR) path, specified in the Advanced Intermediate representation with Extensibility (AIRE) standard. AIRE uses a collection of object instances linked by pointers to represent design information. These objects represent analyzed, elaborated, and executable instances of specific design information (in VHDL or Verilog). In very general terms, the collection of objects represents a very generalized abstract syntax tree, while methods associated with the classes (and thus objects) represent an integrated application programming interface.

SystemC / C++ code was produced from this intermediate format. Professors L.M.Ayough, A.H.Abutalebi, O.F.Nadjarbashi and S.Hessabi developed Verilog2SC, at the University of Technology, Iran. Verilog2SC is an application that converts synthesizable Verilog to SystemC. Dr. Wilson Snyder is currently developing the *Verilator* tool that would produce both C++ and SystemC output from synthesizable Verilog code. Some of these tools are discussed below.

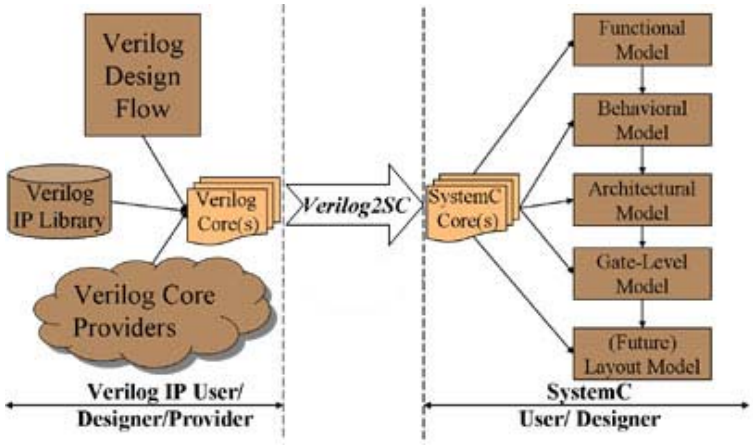

Figure 5 Flow diagram of the Verilog to SystemC tool (Ascend Design Automation)

2.1 HDL to SystemC converters

2.1.1 Verilog2SC

This is a tool developed by the professors at the University of Technology in Iran. It converts the synthesizable subset of the Verilog code to SystemC, Figure 5.

The methodology is based upon compiling the Verilog files into the AIRE hierarchy of classes and implementing the Verilog to SystemC on these classes. The developers of this tool claim to have successfully converted and tested the PicoJava CPU from SUN Microsystems and the XSOC system on a chip. The XSOC project consisted of a total of 1400 lines of code in Verilog which was transformed into 3551 lines of code in SystemC. The simulation time in Verilog was 4.0 seconds and 2.91 seconds in SystemC [20].

This tool is the first commercial Verilog to SystemC converter and is

available for purchase from Ascend Design Automation.

2.1.2 VHDL to SystemC – University of Tübingen

Another tool that converts VHDL to SystemC was developed by Wilhelm-Schickard Institute for Computer Science at the University of Tübingen and is available at the European SystemC users group website. The tool is available for both Linux and Solaris platforms.

2.1.3 VHDL to SystemC – University of Verona

Professor Franco Fummi' s VHDL to SystemC converter is implemented on top of the SAVANT environment developed by Professor Philip Wisley of the University of Cincinnati, see Figure 6.

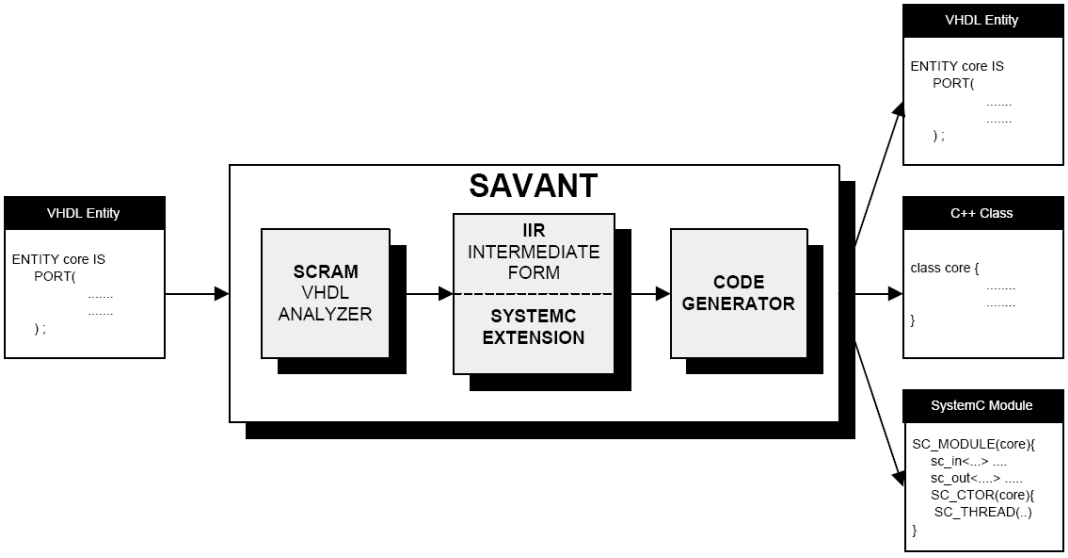

Figure 6 VHDL to SystemC translator (Professor Franco Fummi)

The SAVANT project builds an extensible, object-oriented intermediate format for VHDL. The SAVANT VHDL'93 analyzer covers the entire VHDL'93 language. SAVANT builds the AIRE from VHDL input. The SAVANT tool set also includes a C++ code generator capable of generating C++ from VHDL'93. Translation rules for all the considered VHDL constructs are inserted in a database of rules and implemented in the VHDL to SystemC converter [6].

2.1.4 Verilator

Verilator is a free tool that translates synthesizable Verilog into C++ or SystemC code. *Verilator* supports the synthesis subset of Verilog, initial statements, blocking / non-blocking assignments, tasks and functions. *Verilator* was developed on RedHat Linux 7.2 but runs on any system with GCC and Perl with minor modifications [32]. *Verilator* first converts the files to an intermediate format called the System-Perl format from where the SystemC files are generated.

2.2 VHDL to Verilog

This is a open-source tool created by Ocean Logic, a small intellectual property company in Australia. It translates a subset of RTL VHDL into Verilog. The company develops IP cores for JPEG encryption, motion estimation, discrete cosine transform, inverse discrete cosine transform and Huffman decoders. Developers have successfully translated JPEG, triple DES, AES and MPEG-4 from VHDL to Verilog using this tool. This was written using LEX and YACC, LEX to

split the source file into tokens and YACC to find the hierarchical structure of the program[37].

2.3 Code comparison of VHDL, Verilog and SystemC

In this section a comparison of the syntax and semantics of some of the most basic and widely used statements in VHDL, Verilog and SystemC is made. These comparisons are shown in Figures 7, 8, 9, 10 and 11. In an ideal case the statements shown here in VHDL must be converted to the statements shown here in Verilog and SystemC producing the most efficient code after translation. A similar one to one translation block is not always available due to some structures being supported in one language and not in the other.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity my_model  is
  port(
  input1: in STD_LOGIC;
  input2: in STD_LOGIC;
  output1: out STD_LOGIC;
  output2: out STD_LOGIC;
  );
end my_model;

architecture my_arch of my_model is
begin
  process( input1, input2)
              variable my_var1,my_var2: STD_LOGIC;
  begin
  my_var1 :=  not input1;
  my_var2 :=  not input2;

  output1 <= input1 and my_var2;
  output2 <= input2 and my_var1;

  end process;
end my_arch;
```

Figure 7 Basic structure of a program in VHDL

```
module my_model (input1, input2, output1, output2);

input input1,input2;
output output1,output2;
reg my_var1, my_var2 ;

always @ (input1 or input2)
  begin
        my_var1 =  ~ input1;
        my_var2 =  ~ input2;
        output1 = (input1 and my_var2);
        output2 = (input2 and my_var2);
  end
endmodule
```

Figure 8 Basic structure of a program in Verilog

```
#include "systemc.h"

SC_MODULE (my_model)
{
             sc_in<sc_logic> input1;
             sc_in<sc_logic> input2;
             sc_out<sc_logic> output1;
             sc_out<sc_logic> output2;

            SC_CTOR  (my_model)
             {
                 SC_METHOD ( process );
                 sensitive << input1 << input2;
             }
             void process( )
              {
                     sc_logic my_var1, my_var2;

                      my_var1 = ~input1;
                      my_var2 = ~input2;
                      output1 = input1 & my_var2;
                      output2 = input2 & my_var1;
               }
    };
```

Figure 9 Basic structure of the program in SystemC

VHDL

```
process (input1, input2)
```

In case of a positive edge trigger

```
process(clk)
. . .
if ( clk'event and clk = '1') then
     . . .
```

In case of a negative edge trigger

```
process(clk)
. . .
if ( clk'event and clk = '0') then
     . . .
```

Verilog

```
always @ (input1 or input2)
```

In case of a positive edge trigger

```
always @ (posedge clk) begin
```

In case of a negative edge trigger

```
always @ (negedge clk) begin
```

SystemC

```
SC_METHOD ( process );
sensitive << input1 << input2;
```

In case of a positive edge

```
SC_METHOD ( process );
sensitive_pos  << clk;
```

In case of a negative edge

```
SC_METHOD ( process );
sensitive_neg  << clk;
```

Figure 10 Comparison of process and sensitivity lists

VHDL

```
input1 : in STD_LOGIC ;
input2, input3, input4 : in STD_LOGIC ;
output1 : out STD_LOGIC ;
output2 , output3, output4 : out STD_LOGIC ;
input5 : in STD_LOGIC_VECTOR (2 downto 0);
```

Verilog

```
input input1 ;
input input2, input3, input4 ;
output output1 ;
output output2, output3, output4 ;
input [2:0] input5 ;
```

SystemC

```
sc_in<sc_logic> input1;
sc_in<sc_logic> input1,input2,input3;
sc_out<sc_logic> output1;
sc_out<sc_logic> output1,output2,output3;
sc_in<sc_bv<3> > input5;
```

Figure 11 Code comparison of declaration of ports in VHDL, Verilog and SystemC

3. Development of SystemC modules

The backbone of this project is the *Verilator* tool. Installation was first attempted on RedHat Linux 9. Since SystemC 2.0.1 libraries do not support this platform it was then installed on RedHat 7.3 where too it was not successful. *Verilator* was finally installed on RedHat 8.0. The version of *Verilator* installed was 3.110. The latest version that is available at the time of writing this is 3.202. The 3.2 series *Verilator* have a better code optimization than the earlier 3.1 series.

The prerequisites needed for the *Verilator* tool are Perl 5.6.1, C++ compiler, Verilog-Perl, System-Perl and SystemC.

Verilog-Perl is a library intended as a building point for Verilog support in the Perl language. It builds a netlist out of Verilog files and determines the hierarchy of modules. It renames and cross references Verilog symbols. It preprocesses warnings and assertions for any simulator. Included in the Verilog-Perl is a package called 'vrename', a program renaming signals, modules and parameters across many Verilog files at once.

System-Perl is a library intended as a building point for SystemC support in the Perl language. This package provides several major sub-packages. The SystemC::Parser understands how to read SystemC files, and extract tokens and such. SystemC::Netlist builds netlists out of SystemC files. This allows scripts to determine things such as the hierarchy of SC_MODULEs. The netlist database may

also be extended to support other languages. ‘sp_preproc’ provides extensions to the SystemC language, called the System-Perl language. ‘sp_include’ shows a technique for speeding up SystemC compiles using GCC. ‘sp_makecheck’ allows for cleaning up dependency files when dependencies have been removed or changed. Finally, the ‘src’ directory contains useful C++ utilities for simulation, such as changing ‘cout’ to send to both the screen and a file.

Step by step instructions with explanation to install the pre-requisites for *Verilator* and *Verilator* are available in Appendix A.

3.1 Generation of SystemC modules

The flow for the conversion of Verilog files to SystemC and testing the generated SystemC code is shown in Figure 12. The Verilog code is supplied to *Verilator* that converts it to SystemC. The same testbench used for testing the Verilog code is used for testing the SystemC code. Since *Verilator* does not convert non-synthesizable code and testbenchs are non-synthesizable a manual conversion of the testbench code needs to be done.

3.1.1 D- flip-flop

The flip-flop is one of the smaller cores that were converted to SystemC. The flip-flop is a D flip-flop with asynchronous reset. It has 3 one-bit inputs, a clock, an active low reset signal an input and a one-bit output. The Verilog code was first

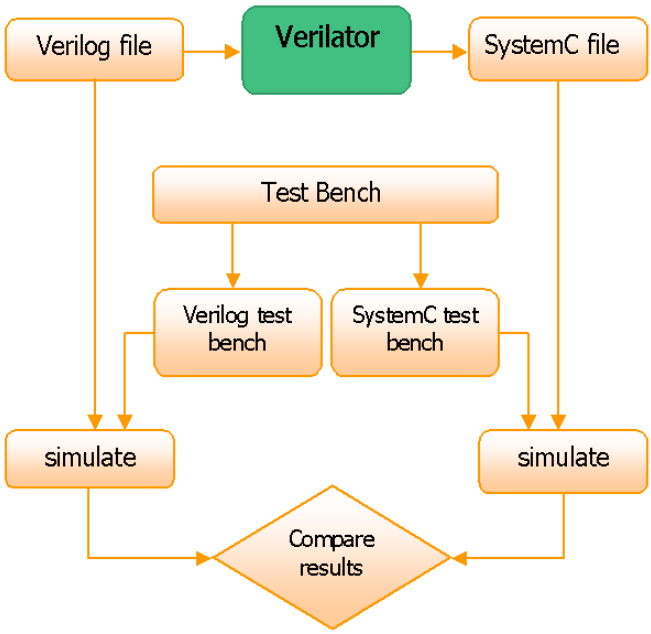

Figure 12 Verification of Verilog to SystemC translation

simulated and the results are displayed in Figure 13. The testbench is comprehensive and tests all aspects of the flip-flop.

To start conversion of the Verilog code into SystemC a directory is created into which the Verilog file is copied. Along with the Verilog file two files vlint and input.vc are also copied into this directory. These two files help in reading all the other files in that directory. The SystemC files are created in a directory called obj_dir after the execution of the *Verilator* commands.

The command that was executed on the top level Verilog file is

```
> perl ../bin/verilator --public --sp –f input.vc top.v
```

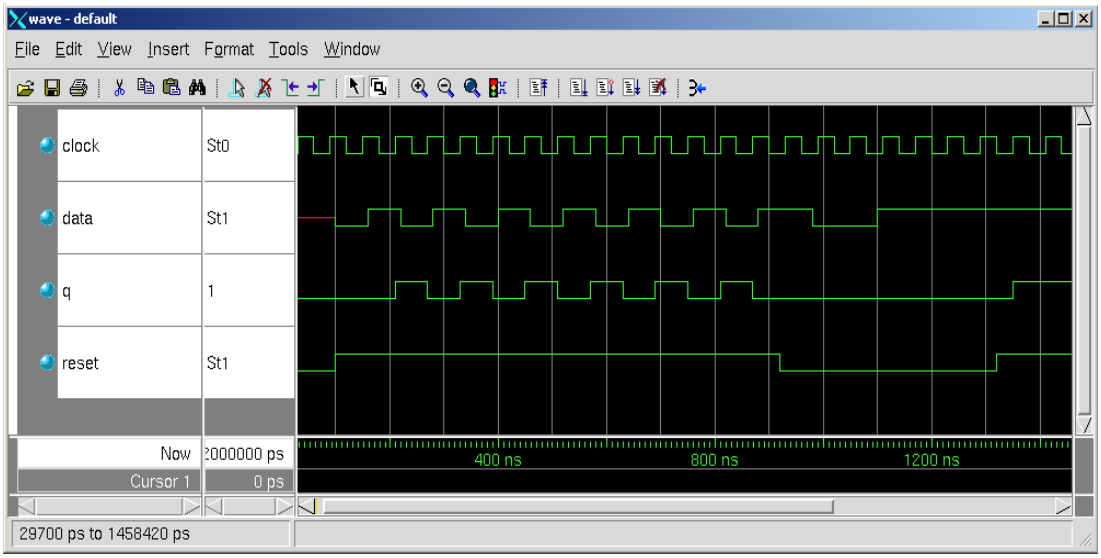

Figure 13 Waveform showing the simulation of the Verilog code of the asynchronous reset D-flip flop

The --public option declares all the signals and modules as public. The --sp option specifies that System-Perl output is desired. The –f option reads the specific input file.

After the execution of this command a directory by name ‘obj_dir’ is created. This directory contains the System-Perl files of the initial Verilog files. These System-Perl files are converted to the SystemC files with the following command.

```
obj_dir >  sp_preproc  --preproc  top.sp
```

This creates the SystemC files in the same directory. This conversion of the flip-flop file created 2 header files, 2 SystemC files and 2 System-Perl files. The System-Perl files are the intermediate format files generated during the conversion process and these files are necessary for the simulation of the generated SystemC

files. The number of files created depends on the number of modules present in the Verilog source files. When the generated SystemC files were simulated using the SystemC-2.0.1 compiler using a similar testbench as that used to test the Verilog code produced similar results.

The simulation waveform of the SystemC code is shown in Figure 14. A value change dump (vcd) file was generated from the testbench for the SystemC simulation. Synopsys' Simwave was used to view the waveform from this dump file. Modelsim was used to simulate the Verilog code and view the waveform.

Figure 15 shows the declaration of ports in Verilog. Figure 16 shows the declaration of ports of the same Verilog file after it was converted to SystemC using *Verilator*.

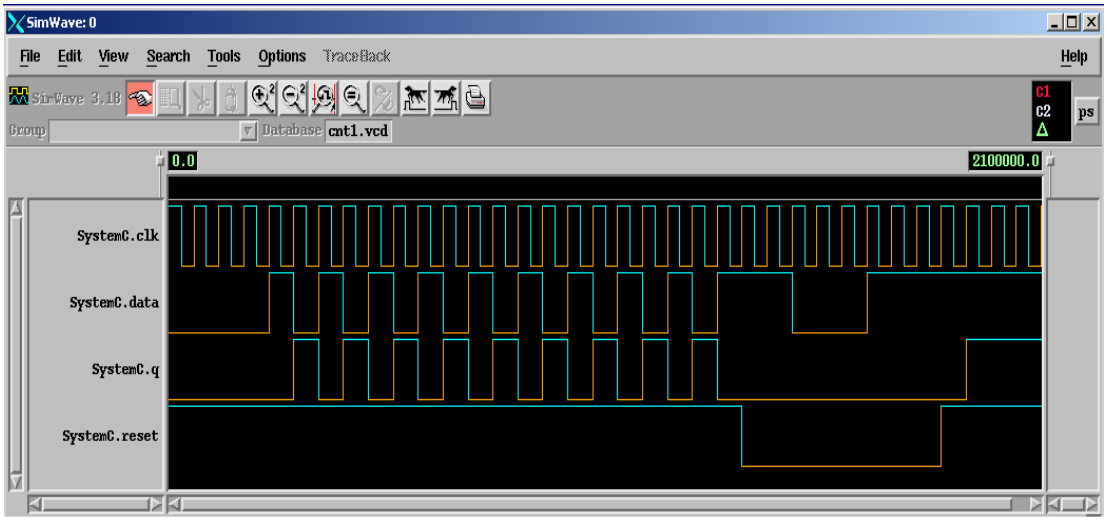

Figure 14 Waveform showing the simulation of the SystemC code of the asynchronous reset D flip-flop

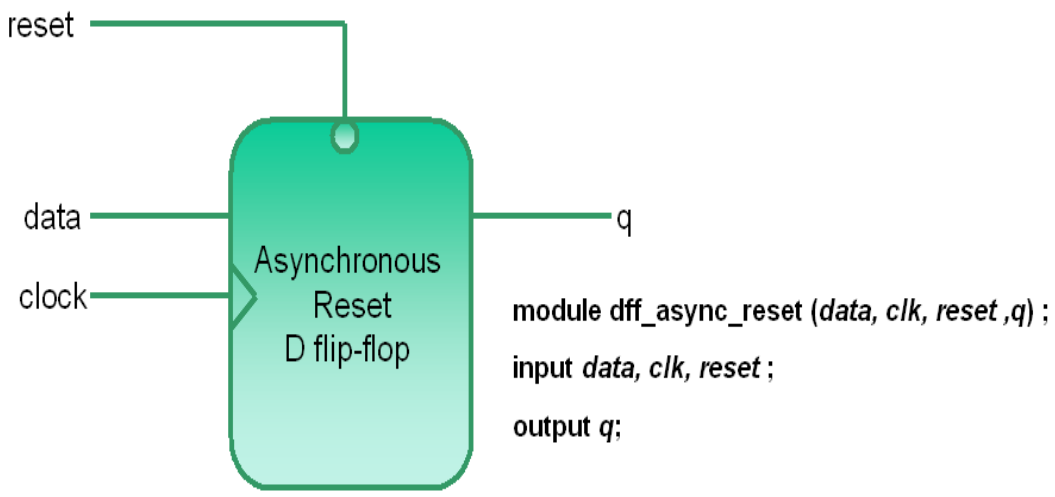

Figure 15 Asynchronous reset D flip-flop (Verilog mode)

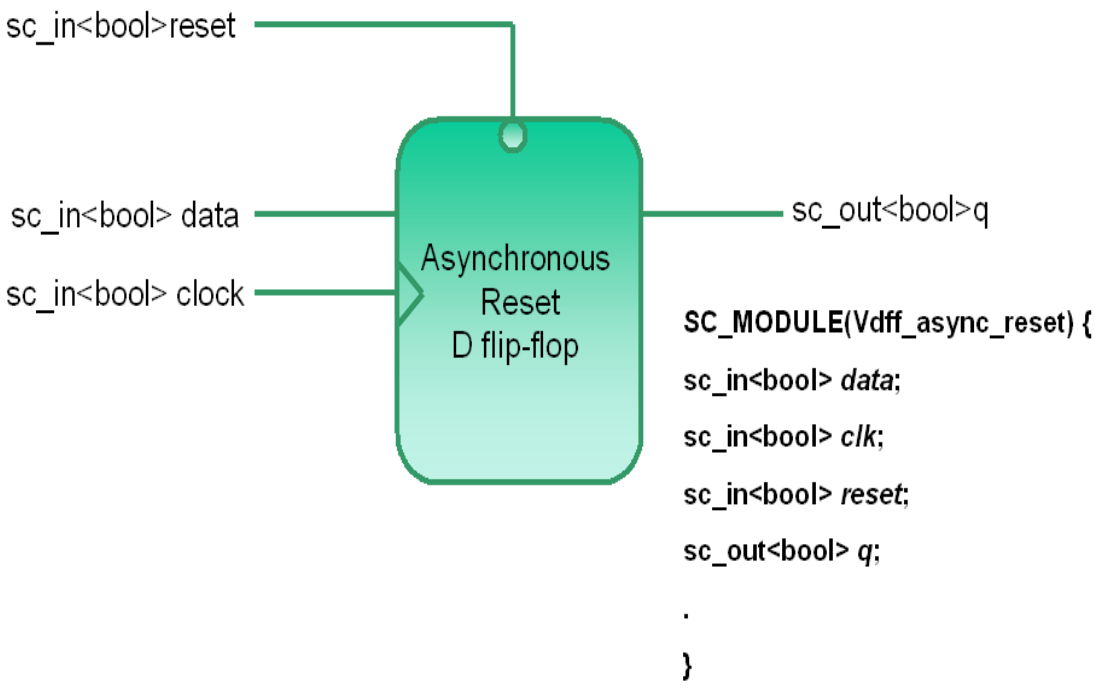

Figure 16 Asynchronous reset D flip-flop (SystemC mode)

3.1.2 Multiplexer

The multiplexer was another smaller core that was converted to SystemC. This multiplexer is a 2 – 1 multiplexer in which the output value is selected from one of the 2 input values.

The multiplexer was first simulated in Verilog. The code consisted of 10 lines. The result of the simulation of the Verilog code is shown in Figure 17. When this was converted to SystemC it produces 2 SystemC files, 2 SystemC header files and 2 System-Perl intermediate format files. All the files are needed for the simulation of the SystemC files. These files were simulated using SystemC-2.0.1 compiler and the wave produced is shown in Figure 18 and 19. A test bench similar to that of the Verilog test bench was used for the simulation of the SystemC files.

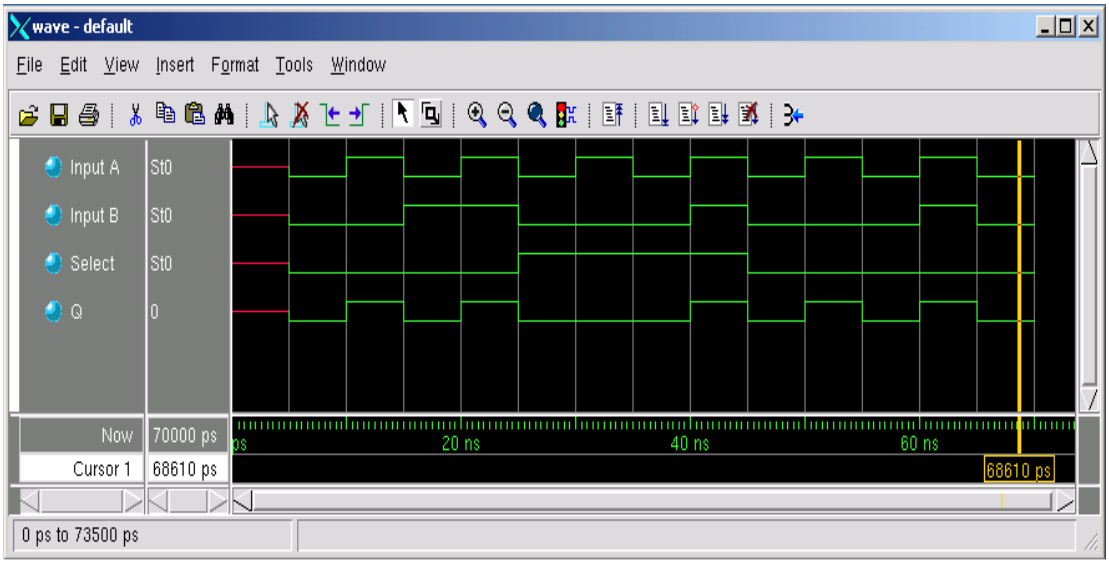

Figure 17 Waveform showing the simulation of the Verilog code of the 2-1 multiplexer

```
            SystemC 2.0.1 --- Sep 15 2003 12:43:46
        Copyright (c) 1996-2002 by all Contributors
                    ALL RIGHTS RESERVED
At time 0 s::Data in = 00  Select = 0 Data out = 0
At time 5 ns::Data in = 10  Select = 0 Data out = 1
At time 10 ns::Data in = 01  Select = 0 Data out = 0
At time 15 ns::Data in = 11  Select = 0 Data out = 1
At time 20 ns::Data in = 00  Select = 1 Data out = 0
At time 30 ns::Data in = 01  Select = 1 Data out = 1
At time 40 ns::Data in = 00  Select = 0 Data out = 0
At time 45 ns::Data in = 10  Select = 0 Data out = 1
At time 50 ns::Data in = 01  Select = 0 Data out = 0
At time 55 ns::Data in = 11  Select = 0 Data out = 1
At time 60 ns::Data in = 00  Select = 1 Data out = 0
```

Figure 18 Screen shot of the command window of the simulation of the SystemC code of the 2-1 multiplexer

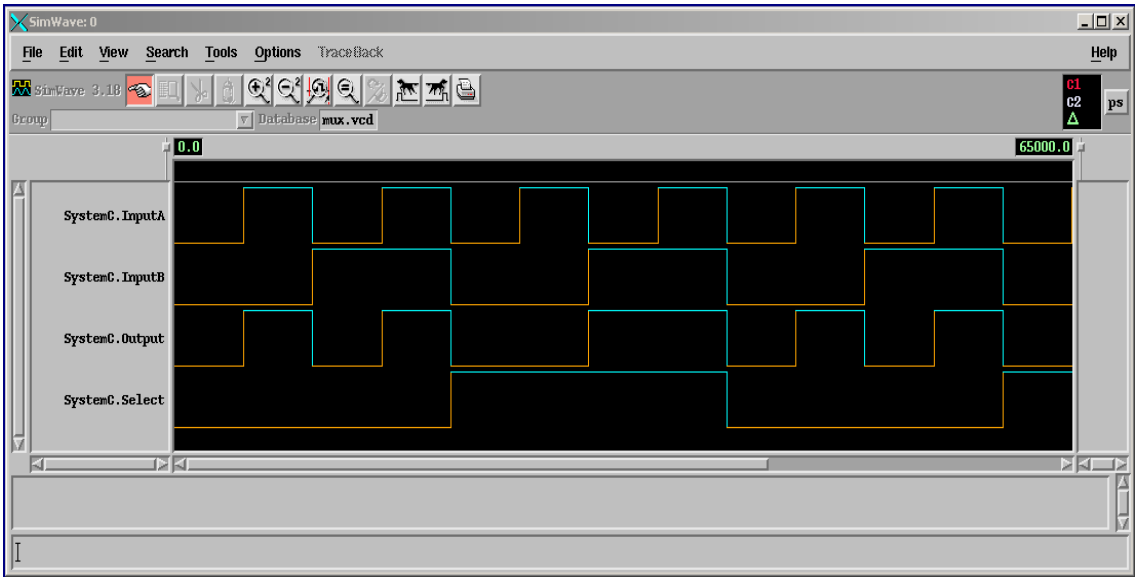

Figure 19 Waveform showing the simulation of the SystemC code of the 2-1 multiplexer

3.1.3 CORDIC

CORDIC is an acronym for CO-ordinate Rotation DIgital Computer. It is a class of shift-add algorithms for rotating vectors in a plane. It performs a rotation using a series of specific rotation angles selected so that each is performed by a shift and add operation. Each stage of iteration produces one bit of accuracy. Rotation of unit vectors provides us with a way to accurately compute trigonometric functions, as well as a mechanism for computing the magnitude and phase angles of input vectors. Vector rotation is also useful in a host of DSP applications like modulation and Fourier transforms. The CORDIC algorithm has also found its way into the 8087 math coprocessor. The CORDIC core here accepts a vector's real component, an imaginary component and an initial phase angle. The CORDIC algorithm iteratively rotates the input vector to the positive real axis, accumulating the angle as it goes

The CORDIC was the biggest core that was converted to SystemC. The code in Verilog was a total of 993 lines and when this was converted to SystemC a total of 17 header files, 17 SystemC files were created along with some intermediate format System-Perl files. The waveform for the simulation of the Verilog file is shown in Figure 20 with binary values, and in Figure 21 with hexadecimal values. The waveform for the simulation of the SystemC is shown in Figure 22 with hexadecimal values. For the sake of ease in observing the waveform a simulation with only one set of values is shown here. The CORDIC has 6 inputs; they are clock, reset, enable imaginary co-ordinate, real co-ordinate and angle, the last three inputs are eight-bit values.

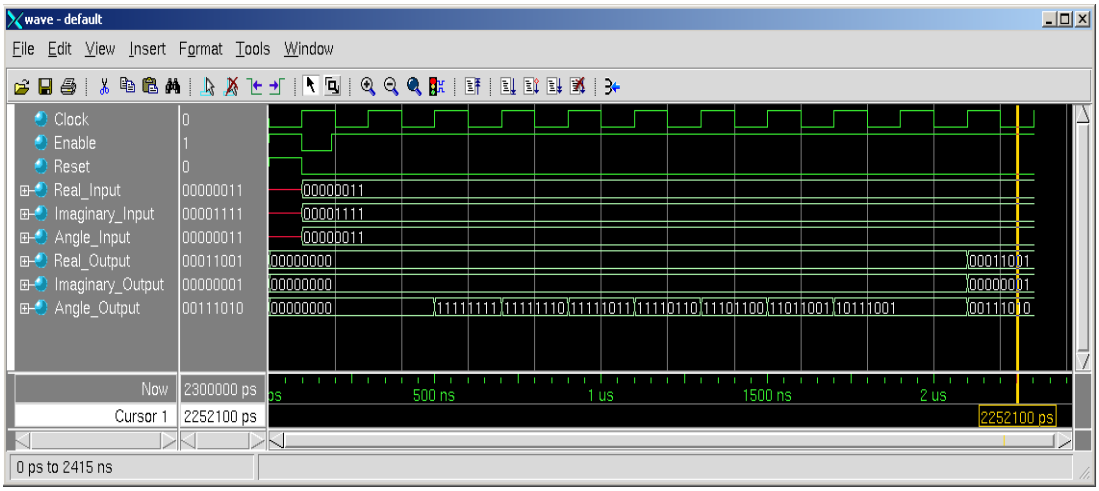

Figure 20 Waveform showing the Verilog code simulation results in binary format of the CORDIC algorithm.

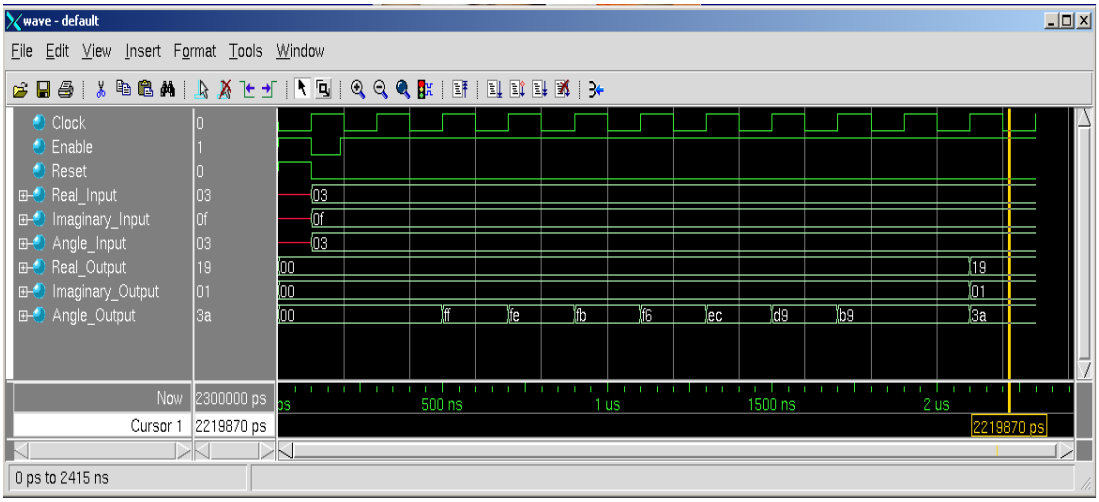

Figure 21 Waveform showing the Verilog code simulation results in hexadecimal format of the CORDIC algorithm

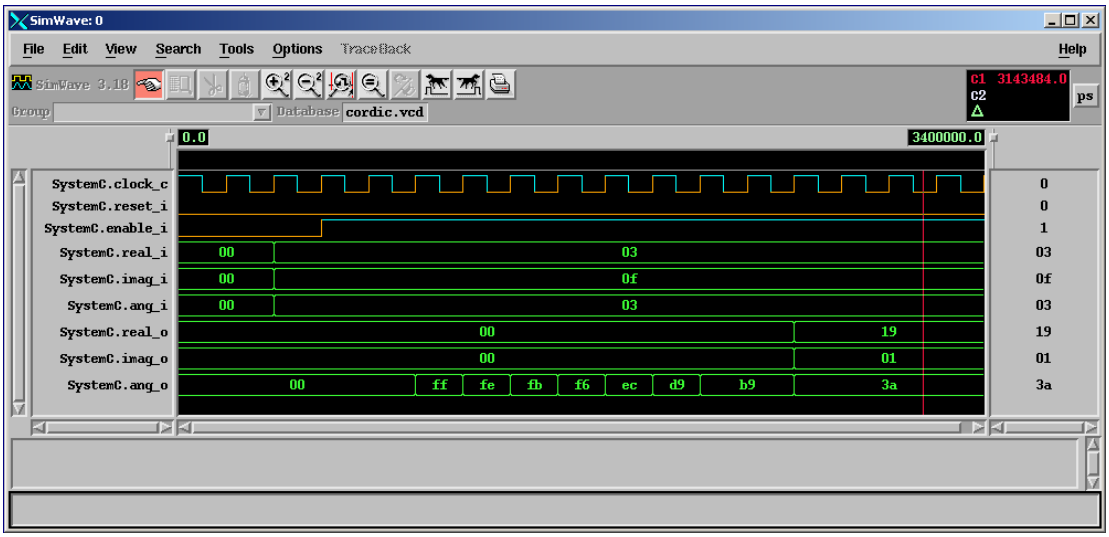

Figure 22 Waveform showing the SystemC code simulation results in binary format of the CORDIC algorithm

As previously mentioned the CORDIC has six inputs of which three are single-bit and three and 8-bit and has three 8-bit output pins as shown in Figure 23.

When *Verilator* was run on this program the single bit pins get converted to sc_in<bool> and sc_out<bool> which is correct but the 8-bit pins get converted to sc_in<uint32_t> and sc_out<uint32_t> instead of getting converted to sc_in<sc_int<8>> and to sc_out<sc_int<8>> as shown in Figure 24. This modification in the top header file was done manually after the conversion, see Figure 25. This modification has to be done because pins 2-32 bits wide become ‘uint32_t’s and pins of a single bit become ‘bool’, unless they are marked with `systemc_clock’, in which case they become sc_clock's. Larger pins become sc_bv's. This was the only alteration that needed manual interference.

```
input  clock_c;
input  enable_i;
input  reset_i;
input  [7:0] real_i;
input  [7:0] imag_i;
input  [7:0] ang_i;
output [7:0] real_o;
output [7:0] imag_o;
output [7:0] ang_o;
```

Figure 23 CORDIC port declarations in Verilog

```
sc_in<bool> clock_c;
sc_in<bool> enable_i;
sc_in<bool> reset_i;
sc_in<uint32_t>     real_i;
sc_in<uint32_t>     imag_i;
sc_in<uint32_t>     ang_i;
sc_out<uint32_t>    real_o;
sc_out<uint32_t>    imag_o;
sc_out<uint32_t>    ang_o;
```

Figure 24 CORDIC port declarations by Verilator

```
sc_in<bool> clock_c;
sc_in<bool> enable_i;
sc_in<bool> reset_i;
sc_in<sc_int<8> >   real_i;
sc_in<sc_int<8> >   imag_i;
sc_in<sc_int<8> >   ang_i;
sc_out<sc_int<8> >  real_o;
sc_out<sc_int<8> >  imag_o;
sc_out<sc_int<8> >  ang_o;
```

Figure 25 CORDIC port declarations after correcting Verilator produced declarations

3.1.4 RISC processor

This is an 8-bit RISC microcontroller available at 'The Free IP Project' webpage. www.free-ip.com. This core has a total of 1680 lines of code. When it was attempted to convert this core to SystemC several limitations of the *Verilator* tool were exposed. The first hurdle encountered was at the instance of the occurrence of the casex statement shown in Figure 26. 'casex' and 'casez' are variations of the case statement. In a 'casez' statement, the value of 'z' that appears in the case expression and in any case item expression is considered as don't–care, that is, that bit is ignored.

In a casex statement, both the values 'x' and 'z' are considered as don't cares. *Verilator* wanted 'casex' with 'x' to be replaced with 'casez' with '?'. When 'casex' with 'x' were replaced at all the instances with 'casez' with '?' this error was no longer encountered.

```
always @(inst) begin
   casex (inst)
      12'b0000_0000_0000: inst_string = "NOP     ";
      12'b0000_001X_XXXX: inst_string = "MOVWF   ";
      12'b0000_0100_0000: inst_string = "CLRW    ";
      12'b0000_011X_XXXX: inst_string = "CLRF    ";
      12'b1110_XXXX_XXXX: inst_string = "ANDLW   ";
      12'b1111_XXXX_XXXX: inst_string = "XORLW   ";

      default:            inst_string = "-XXXXXX-";
   endcase
end
```

Figure 26 Casex code instant

On rerunning *Verilator* after the ‘casex’ statements were replaced with the ‘casez’ statements another error was encountered when a signal was declared with name ‘do’. This was a conflict as ‘do’ was a reserve word in C. When the name of this signal was changed this error no longer occurred.

The next hurdle encountered was at an instance of the code when a comparison of two signals was carried out. One of the signals was 9 bits wide and the other signal was either 5 bits or 9 bits depending on the situation. Since modifying this required a much deeper understand of the code and will finally lead to a complete modification of the code it was no longer attempted to convert the RISC microcontroller code to SystemC. The screen shot of the error displayed is shown in Figure 27.

```
[root@grdsl-42 test_sp]# perl /home/sidd/v2sc/verilator-
3.110/test_sp/../bin/verilator              --public --sp -f
/home/sidd/v2sc/verilator-3.110/test_sp/../test_v/input.vc cpu.v
%Warning-WIDTH: ../test_v/cpu.v:334: Operator EQ expects 5 bits on the
RHS, but RHS's VARREF generates 3 bits.
%Warning-WIDTH: Use "/* verilator lint_off WIDTH */" and lint_on around
source to disable this message.
%Warning-WIDTH: ../test_v/cpu.v:845: Operator ASSIGNDLY expects 11 bits
on the Assign RHS, but Assign RHS's CONST generates 9 bits.
%Error: Exiting due to 2 warning(s)
%Error: Command Failed /home/sidd/v2sc/verilator-
3.110/bin/../verilator_bin -MMD -Mdir obj_dir --sp --public --l2name --
prefix Vcpu --mod-prefix Vcpu -f obj_dir/Vcpu.flags_vpp  cpu, 6, stopped
at /home/sidd/v2sc/verilator-3.110/test_sp/../bin/verilator line 310.
[root@grdsl-42 test_sp]#
```

Figure 27 Screen shot of the error at the ‘bit width not same’ instance of the RISC code

3.2 Limitations of Verilator

- *Verilator* supports only the Synthesis subset with a few minor additions such as $stop, $finish and $display.
- As a 2 state translator, tristate and inouts are not supported.
- All variables in *Verilator* are unsigned, including integers.
- The division and modulus operators are limited to 32 bits.
- You cannot have local variable declarations in begin/end blocks.
- When initializing an array, you need to use non-delayed assignments.

3.3 Inference on Verilator

Verilator does a good job with synthesizable Verilog code but fails in certain conditions. The SystemC code generated by *Verilator* can be used for simulation purposes alone. The code produced by *Verilator* is inefficient for synthesis and platform dependent. It needs the SystemPerl package to be installed for simulating code generated by *Verilator*.

3.4 Synthesis of SystemC code

SystemC Compiler synthesizes SystemC RTL modules or a design with integrated RTL and behavioral modules into a gate-level netlist. It can also synthesize a SystemC behavioral module into RTL or a gate-level netlist, see Figure 28. This netlist can be used as input to other Synopsys tools such as Design Compiler

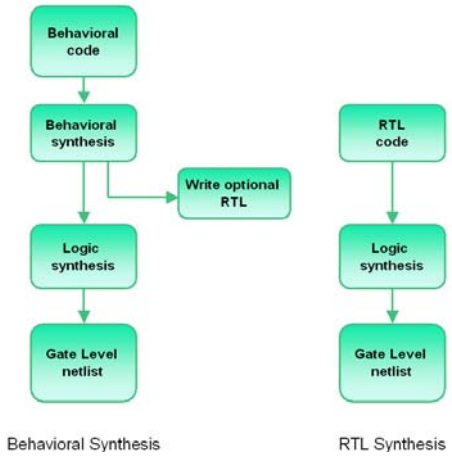

Figure 28 Synthesis with SystemC compiler

and Physical Compiler after synthesis.

SystemC Compiler requires a SystemC RTL description, a technology library, and a synthetic library. Figure 29 shows the flow into and out of SystemC Compiler. The RTL description is independent of the technology and using SystemC Compiler, the target technology library is specified. For this synthesis TSMC.18 technology was used. SystemC compiler was used to synthesis a 16-bit counter. An area report was generated showing the total area of the design, the number of nets and the number of cells using the TSMC.18 technology, see Figure 30.

The generated Verilog code was simulated and the simulation results were compared with the results of the original SystemC code. The waveforms are as expected and the generated Verilog code did not require any corrections to be made. The SystemC and the Verilog simulation waveforms are shown in Figures 31 and 32.

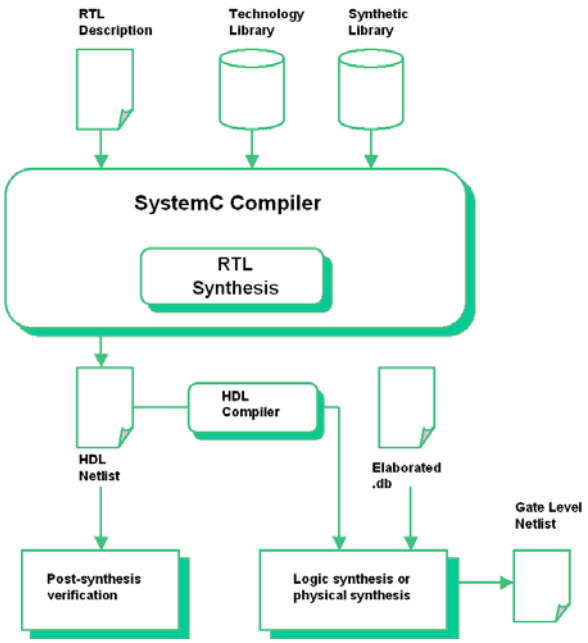

Figure 29 SystemC compiler input and output flow for RTL synthesis

```
Information: Updating design information... (UID-85)
Automatic time-borrowing...

****************************************
Report : area
Design : cnt
Version: 2003.06-SP1
Date   : Thu Apr  1 12:05:48 2004
****************************************

Library(s) Used:

    typical (File: /sw/CDS/ARTISAN/TSMC18/aci/sc/synopsys/typical.db)

Number of ports:                6
Number of nets:                35
Number of cells:               31
Number of references:          18

Combinational area:        455.716827
Noncombinational area:     369.230408
Net Interconnect area:       0.282013

Total cell area:           824.947205
Total area:                825.229309
```

Figure 30 Area report

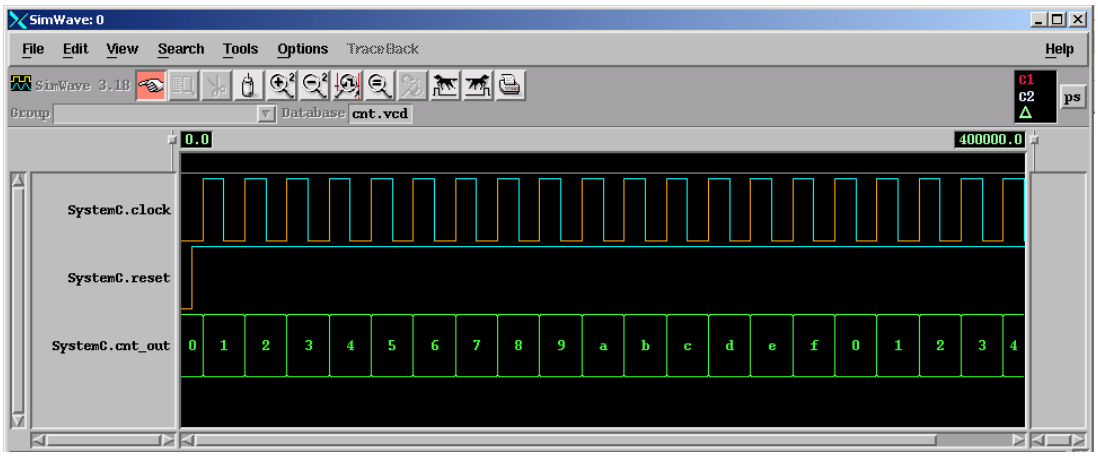

Figure 31 Simulation waveform of the SystemC code of a 16 bit counter

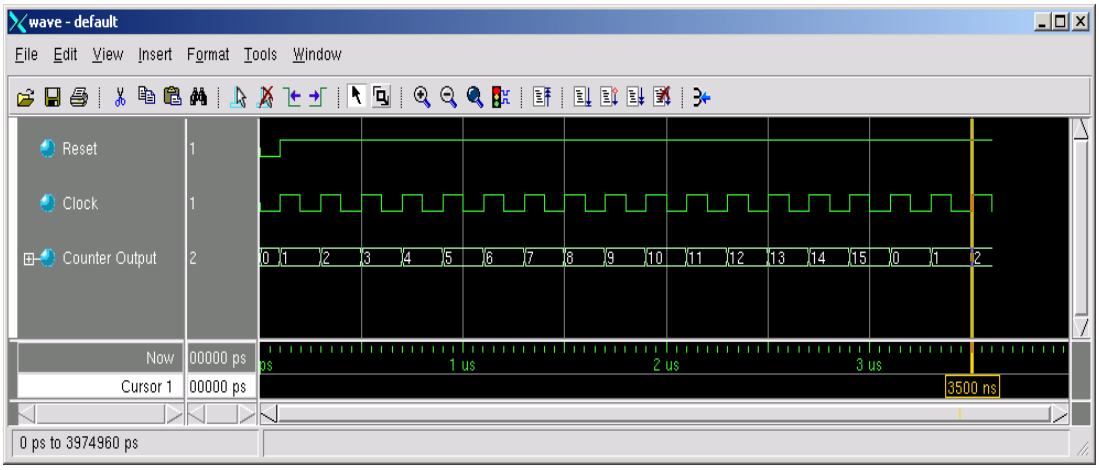

Figure 32 Simulation waveform of the Verilog code generated by DC Shell from the System C code of the 16-bit counter

The following commands were used in DC Shell to synthesize the counter.

```
compile_systemc  -rtl  -rtl_format verilog “cnt.cpp”
compile_systemc -rtl -rtl_format db "cnt.cpp"
create_clock clock -p 10 -name "clock"
compile -map_effort medium
write -hier -f db -o cnt.db
report_area > cnt_area.rpt
```

3.4.1 Synthesis of Verilator generated SystemC code

Synthesis of *Verilator* generated SystemC code is not synthesizable as the generated code is not platform independent. The simulation of the *Verilator* generated SystemC code requires the intermediate format SystemPerl files and a few *Verilator* files. *Verilator* is currently not available for the Solaris platform where in the synthesis tools are available. Attempts to port *Verilator* onto Solaris during this research were not successful.

Furthermore the developer of *Verilator* believes the SystemC code generated by *Verilator* would not make for efficient synthesis in its current stage of evolution.

This will not bother us as the idea for developing SystemC modules is for reducing the time of simulation of the modules and not to synthesize it. Several researchers have proven SystemC simulates up to 3 times faster than VHDL and Verilog [24] [20] but VHDL and Verilog produce the best synthesis results. The best approach would be to use SystemC for simulation and Verilog and VHDL for synthesis.

4. Translating between HDLs

A translation tool between HDLs was necessary in this research to convert cores available in VHDL to Verilog since *Verilator* only converts cores available in Verilog to SystemC. The generation of Verilog from VHDL (or vice versa) is not always possible because of the different constructs supported by one language and not the other. Anyhow in the latest flavors of Verilog (2001 and SystemVerilog) these differences are reduced, as Verilog tends to catch up with VHDL. A human designers 'clean up' might be needed while using these converters.

This research investigated the VHDL to Verilog converter developed by Ocean Logic, an intellectual property company in Australia and Mentor Graphic's HDL Designer.

4.1 Verilog to VHDL conversion using MentorGraphics' HDL Designer

HDL Designer is a tool for creating and managing complex Verilog, VHDL, and mixed-language ASIC and FPGA designs. It enables the design simulation, debugging and synthesis of a complete HDL design.

A picture is worth a million words and that's why most engineers create a graphical diagram of their design. HDL designer series automatically generates diagrams from HDL code in block diagram, state machine and flow chart formats. HDL Designer series also provides a facility where the design can be entered in

graphics format, flow chart, block diagram or state machine and it generates the HDL code from this graphic. This was the feature that was made use of to generate Verilog code from VHDL. The VHDL code was first loaded and the block diagram of the VHDL code was generated. This block diagram was used to generate the Verilog code. This method will work as long as there are no embedded blocks inside a block in the block diagram. A human clean up is needed finally to use the generated code. The detailed procedure in the next section will show how this conversion was carried out. The human clean up required and the simulation results are also shown.

4.1.1 Translating HDLs using Mentor Graphics' HDL Designer

- Mentor graphics HDL designer is invoked with the following commands.

```
> mentor_tools
> fpgadvpro &
```

- A 'Getting Started' window opens up as shown in Figure 33. Select 'Create a new project' and click 'OK'.
- After clicking 'OK' a new window, 'Creating a new project' will open. In this window, in the 'name of new project' column enter the name of the project and in the 'Directory in which your project folder will be created' column, enter the folder where the new project is to be created and click 'Next'. Refer to Figure 34.

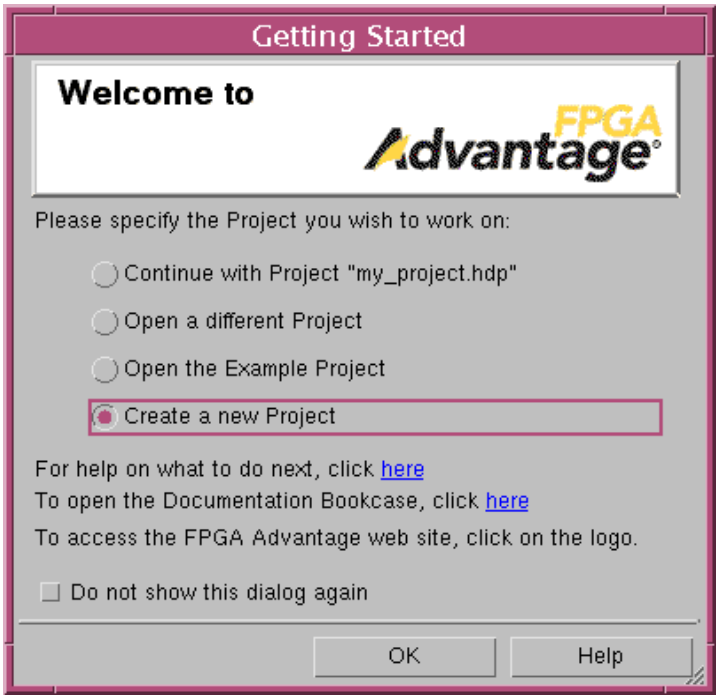

Figure 33 Getting started window

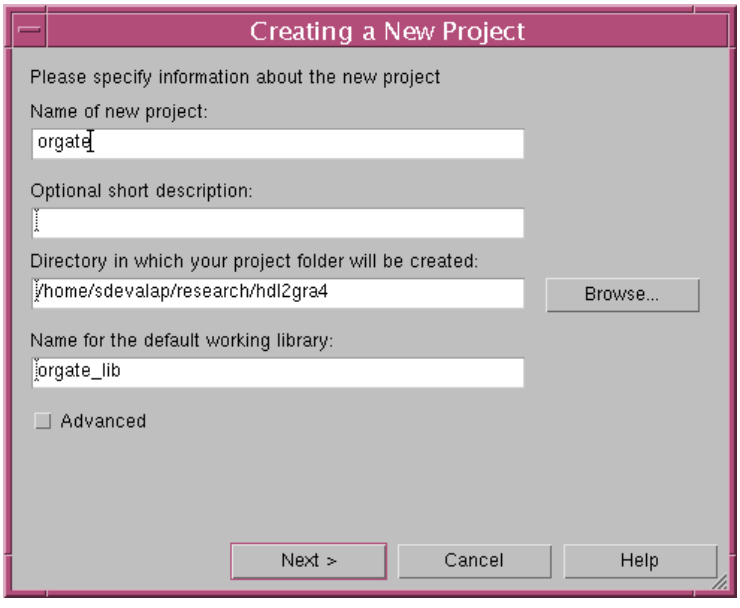

Figure 34 Creating a new project window

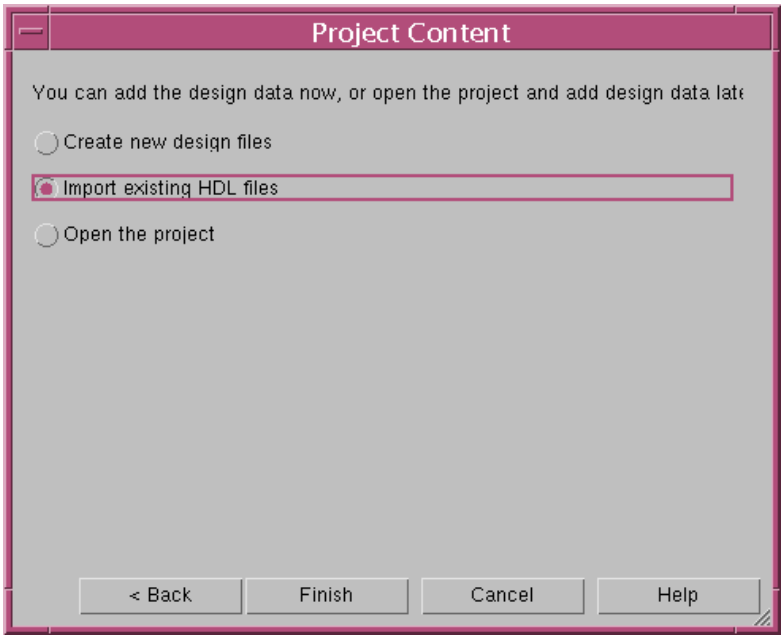

Figure 35 Project content window

- 'Project Summary' window opens, click 'Next'.
- 'Project Content' window opens. Select 'Import existing HDL files' and click 'Finish'. Refer to Figure 35.
- A 'HDL Import Wizard' opens. Select 'Specific HDL' files and 'VHDL' and click 'Next'. Refer to Figure 36.
- A 'HDL Import Wizard – Specific HDL Source Files' window opens. In the 'Directory' column browse to the directory where your VHDL file is located and in the 'Files of Type' column select VHDL files. The VHDL files will show in the 'Files in Directory' window. Highlight the file and click 'Add'. The files will appear in the 'Files to import' window. Click 'Next'. Refer to Figure 37.

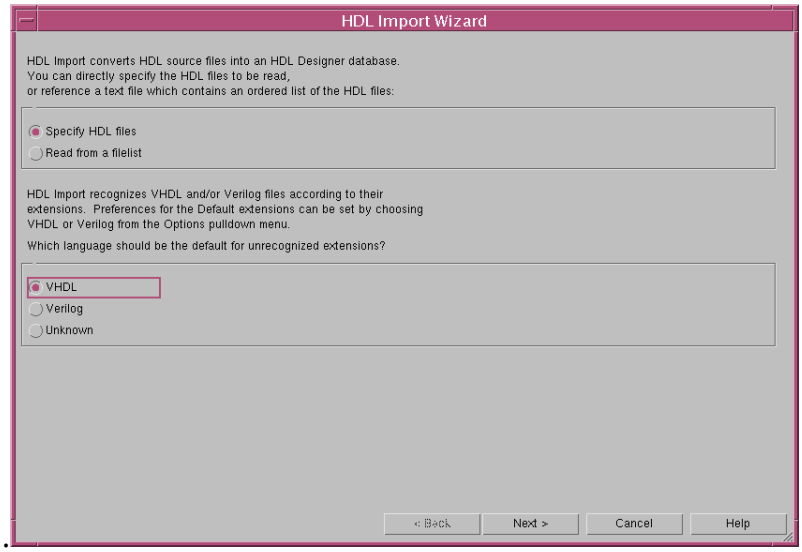

Figure 36 HDL import wizard window

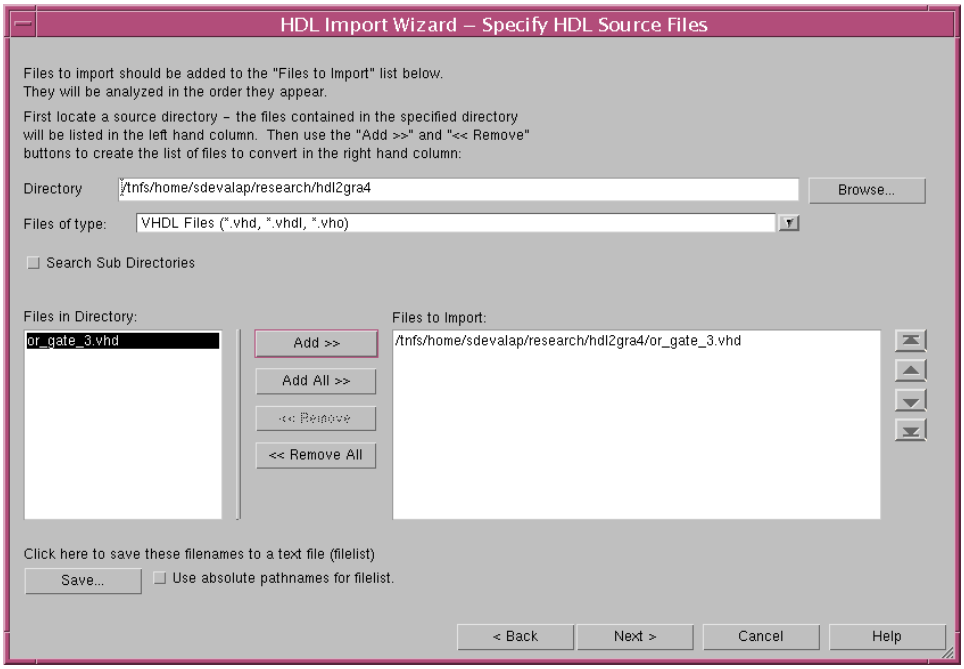

Figure 37 HDL import wizard – specific files window

- ‘HDL Import Wizard – Target Libraries’ window opens. Click ‘Next’.
- ‘HDL import Wizard – Target Directories’ window will open. In the ‘Additional Options’ column only ‘Overwrite existing files’ and ‘Import directory structure’ should be selected.
- In the log window that runs simultaneously ‘1 file imported to 1 library’ is displayed.
- In the ‘Design Manager’ window, expand the project file by clicking on the ‘+’ sign in the ‘Design Explorer’ sub-window.
- In the ‘Design Explorer’ sub-window in the ‘Design Manager’ window select the top architecture file. This is indicated with a blue triangular arrow pointing toward it from the left. Selecting this will enable the ‘Convert to Graphics’ icon in the “Explore’ sub-window. Click on this icon. Refer to Figure 38.

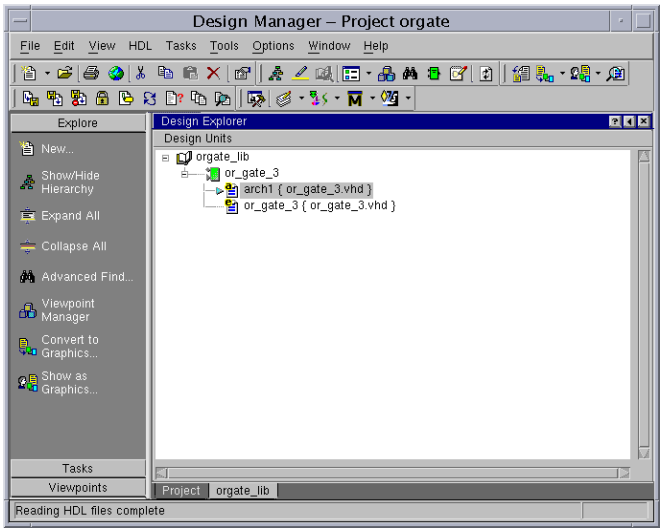

Figure 38 Design manager window

- A ‘Convert to Graphics Wizard – View Styles’ window will open. The ‘Hierarchy Description’ block will be disabled. In the ‘Leaf Level Descriptions’ block select all options and click ‘Next’
- A convert to ‘Graphics Wizard – Advanced’ window will open. Select ‘Overwrite’ and ‘Set Default Views’ options and click ‘Finish’
- In the log window a message indicating the successful creation of the lock diagram is displayed.
- The icon for the block diagram is created in the Design Explorer sub-window. Click on this to display the block diagram.
- The ‘Block Diagram’ window will open showing the block diagram. In this window, select ‘Main’ from the ‘Options’ menu, look at Figure 39.

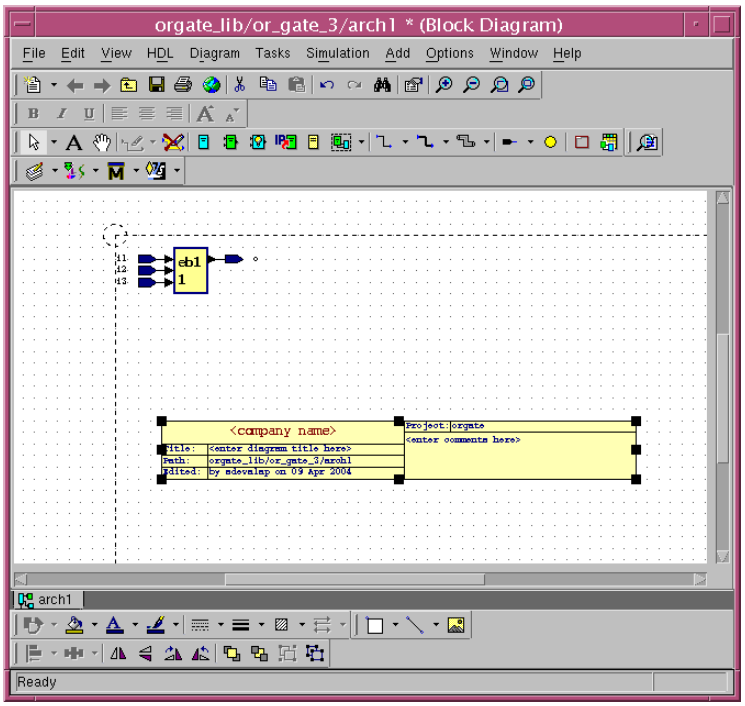

Figure 39 Block diagram window

- ‘Main settings window will open. In the ‘Default language for new views’ section select ‘Verilog’ and click ‘Apply’ and ’OK’. This will force all new block diagram windows opened from now onwards to be Verilog. Refer to Figure 40.
- In the ‘Block Diagram’ window select ‘View generated HDL’ from the ‘HDL’ menu. This will display the VHDL code from which the graphics were generated.
- Select ‘New’ from the ‘File’ menu. In the ‘New’ menu select ‘Graphical View’ and then select ‘Block Diagram’. A new blank ‘Block Diagram’ window will open.

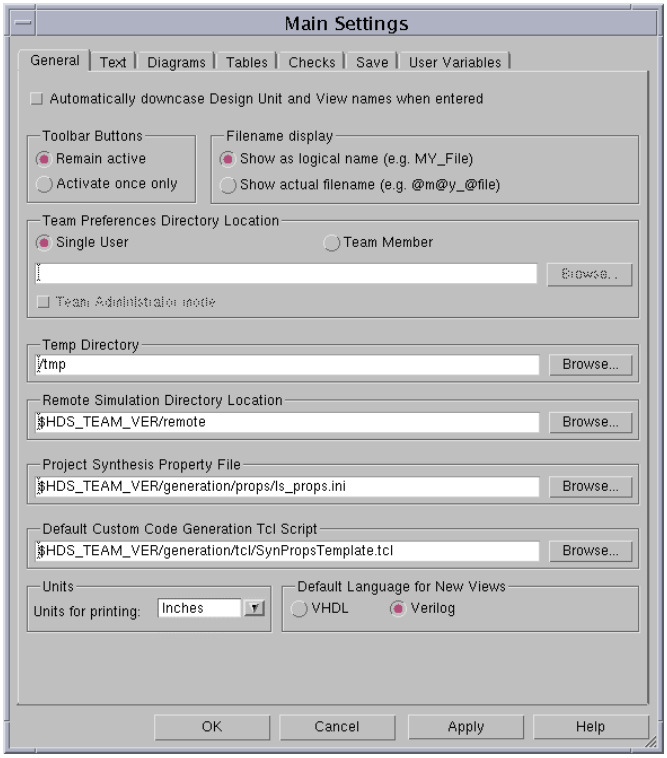

Figure 40 Main settings window

- In the original ‘Block Diagram window select ‘Select All’ from the ‘Edit’ menu and then select ‘Copy from the ‘Edit’ menu.
- Select ‘Paste from the ‘Edit’ menu in the new ‘Block Diagram’ window. This copies the VHDL block diagram from the VHDL ‘Block Diagram’ window to the new Verilog ‘Block Diagram’ window. The original block diagram window can now be closed.
- In the new ‘Block Diagram’ window select ‘Save’ from the ‘File’ menu. A ‘Save as Design Unit View’ window will open. In the ‘Design Unit’ column, replace the name of the existing design unit with a new one and click ‘OK’. Refer to Figure 41.
- In the ‘Block Diagram’ window select ‘Generate’ from the ‘Tasks’ menu and then select ‘Run Single’ from the ‘Generate’ menu.
- Generation completed successfully’ message will be displayed in the ‘Log Window’
- In the ‘Block Diagram’ window select ‘View Generated HDL’ from the ‘HDL’ menu. This will display the generated Verilog code, see Figure 42. The generated Verilog code is also present in the project directory.

In the window where the Verilog code is displayed the lines are marked with a bullet on the left where the conversion was incomplete or where the conversion is not correct. This is the point where a manual conversion or cleaning up is needed. Certain code segments like the porting of a multi-bit input output signals are also not converted correctly but are also not marked with any bullets.

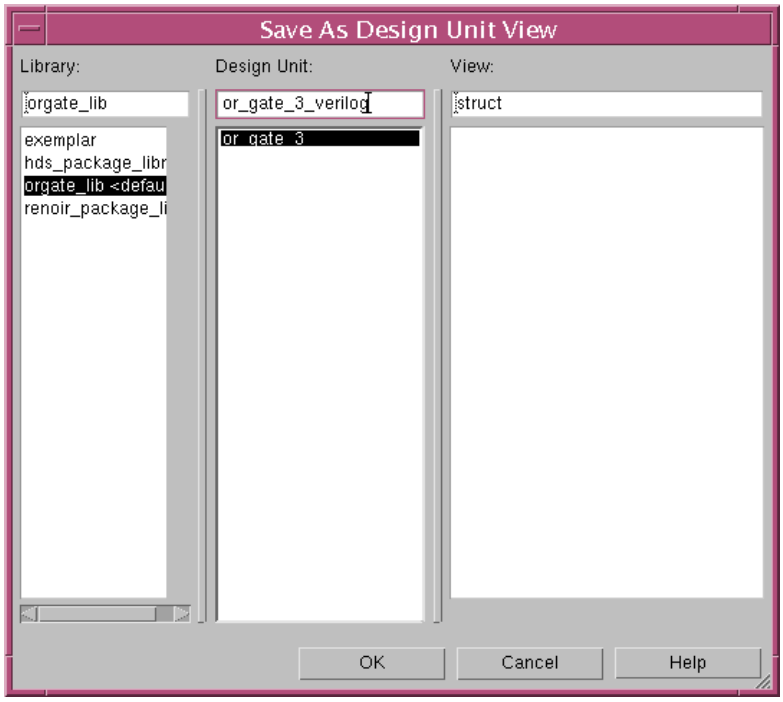

Figure 41 Save as design unit view

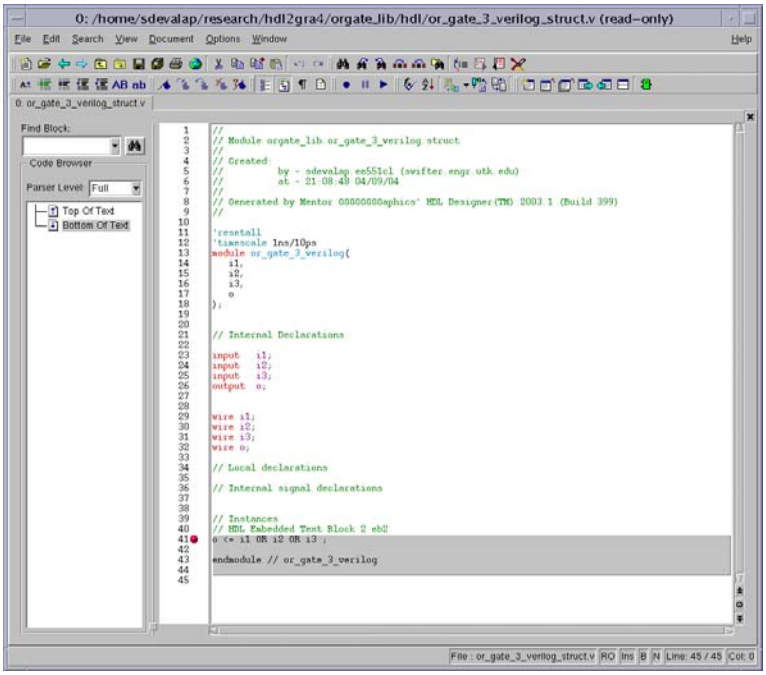

Figure 42 Generated Verilog code

4.1.2 Conversion results

The generated Verilog code was not error free. The Verilog code display window marks the line number where it did not do a correct conversion with a bullet. In this case the conversion of the assignment of the or'd values of the three inputs to the output was not done. Instead of assign o = i1 | i2 | i3, the code remained as o<=i1 OR i2 OR i3. This change was done manually. The simulation of the two codes produced results as expected. The VHDL and Verilog simulation waveforms are shown in Figures 43 and 44 respectively. Gate level back annotation simulations were also performed on the VHDL and the generated Verilog codes. The waveforms are shown in Figures 45 and 46 respectively. The simulation waveforms confirm the functionality of the code.

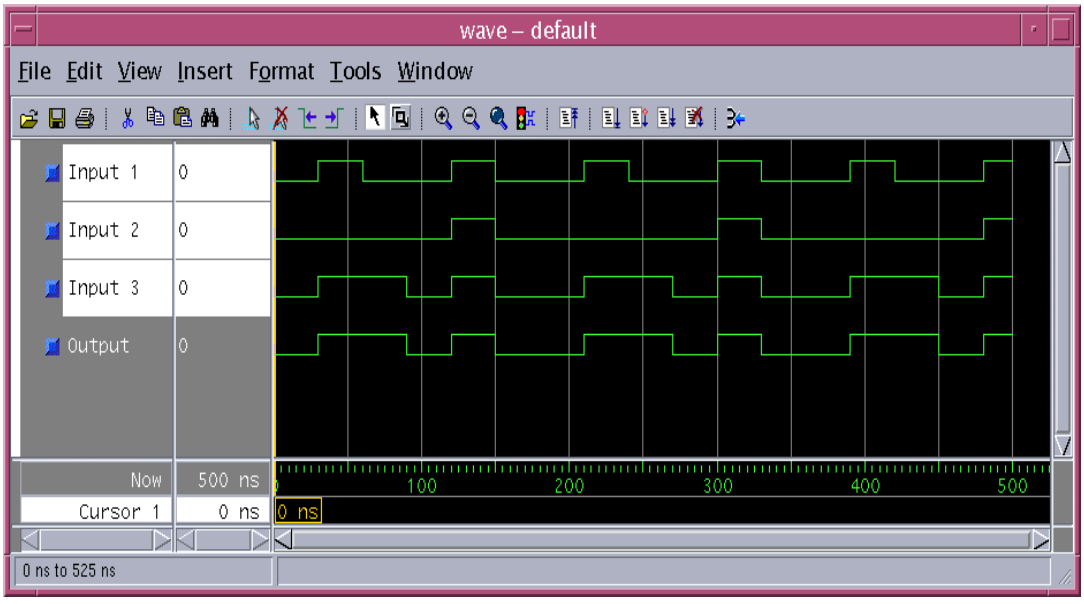

Figure 43 Simulation waveform of the VHDL code of the 3-input OR gate

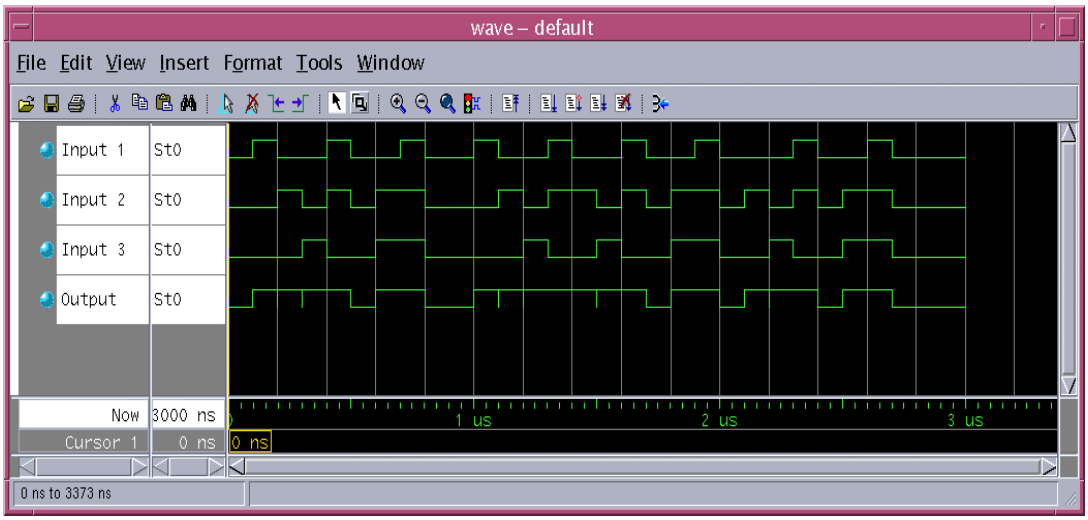

Figure 44 Simulation waveform of the Verilog code generated from VHDL by HDL Designer of the 3-Input OR gate

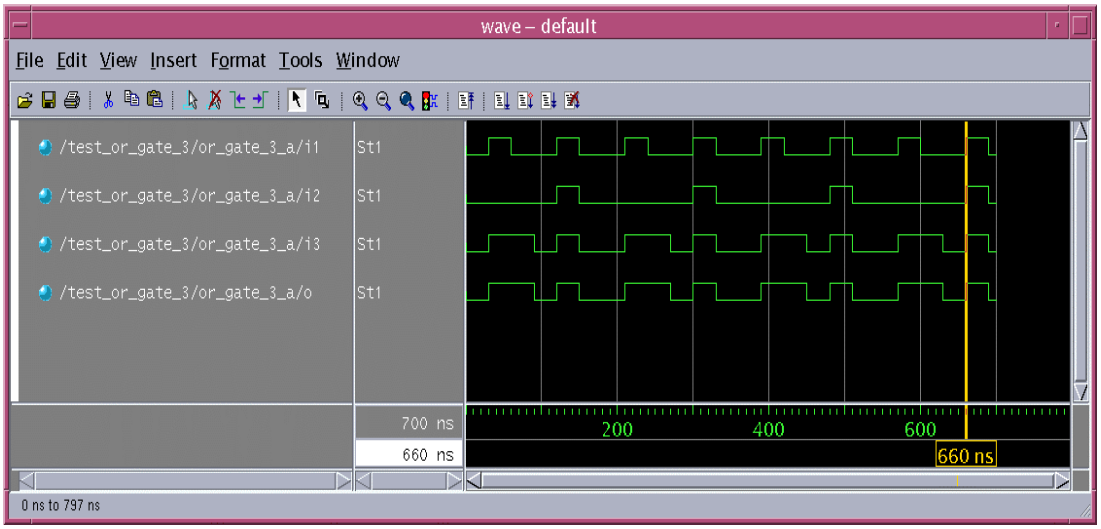

Figure 45 Post layout simulation waveform of the VHDL code of the 3-input OR gate

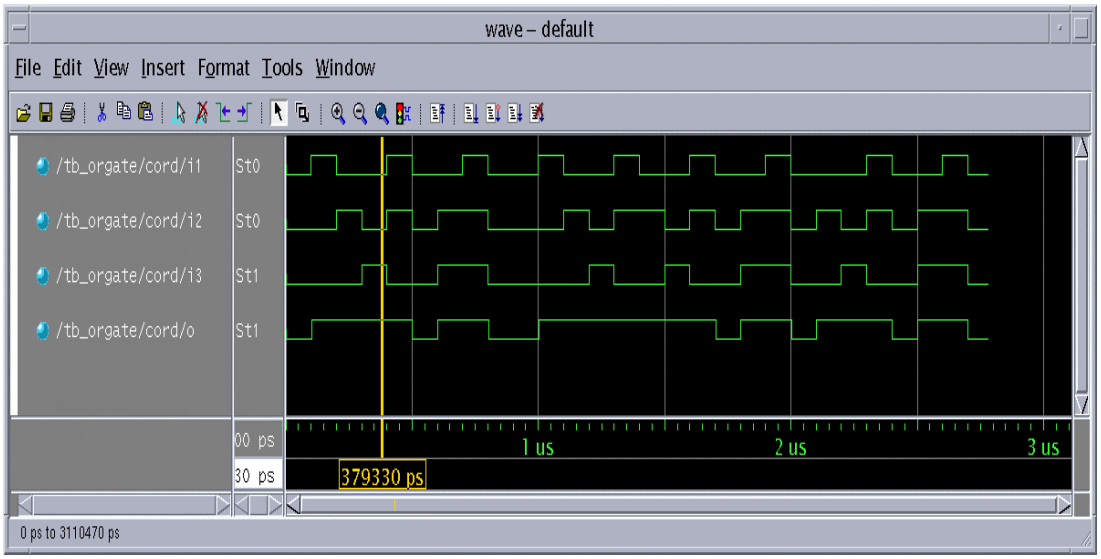

Figure 46 Post layout simulation waveform of the Verilog code generated from VHDL by HDL Designer of the 3-input OR gate

Using Mentor Graphics' HDL Designer the same procedure can be followed to produce VHDL code from Verilog with minor changes in the selection of options.

4.2 Translating VHDL to Verilog using Ocean Logic's VHDL to Verilog converter

'VHDL to Verilog' is a tool developed by a small IP company in Australia called Ocean Logic. The company claims the tool converts the synthesizable RTL subset of VHDL and has successfully converted all IP developed by them (MPEG4, DEC, IDCT) from VHDL to Verilog, This tool was developed mainly for the purpose of converting the IP cores (mentioned above) to Verilog since some of the company's clients wanted the cores in Verilog and VHDL and this tool served this purpose.

4.2.1 Installation and execution

VHDL to Verilog can be downloaded from the company's website at www.ocean-logic.com for no cost. The tool was installed on RedHat 8 with ease. To install the software after downloading and unzipping it, move to the 'src' directory and type 'make'. This will install the software. To use the software type the following command in the src directory.

```
/src > vhd2vl  < source_file.vhd> <desired_name_of_output_file.v>
```

4.2.2 Conversion results

This tool was used to convert a D-Flip Flop, multiplexer and a 3-Input OR gate from VHDL to Verilog and these simulations worked fine. The simulation waveforms for the 3-Input or gate are shown in Figures 47 and 48. When larger cores like the DES available on the Opencores.org website were attempted to be converted to Verilog the tool failed.

To sum up, the 3-Input OR gate code was first written in VHDL. This was converted to Verilog by both Mentor Graphics' HDL Designer and Ocean Logic's VHDL to Verilog converter. The HDL Designer generated code had to be cleaned up but the Verilog code generated by VHDL to Verilog converter was untouched by hand. Post layout simulations were performed on the original and the converted code generated by HDL Designer. The Verilog code generated by VHDL to Verilog converter was then converted to SystemC using *Verilator,* see Figure 49.

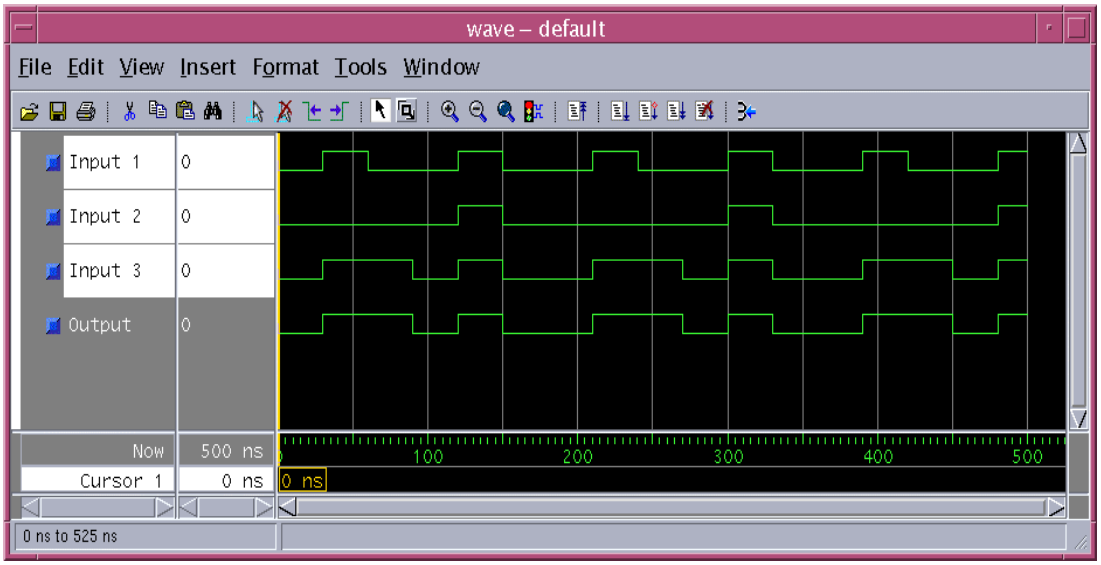

Figure 47 Simulation waveform of the VHDL code of the 3-input OR gate.

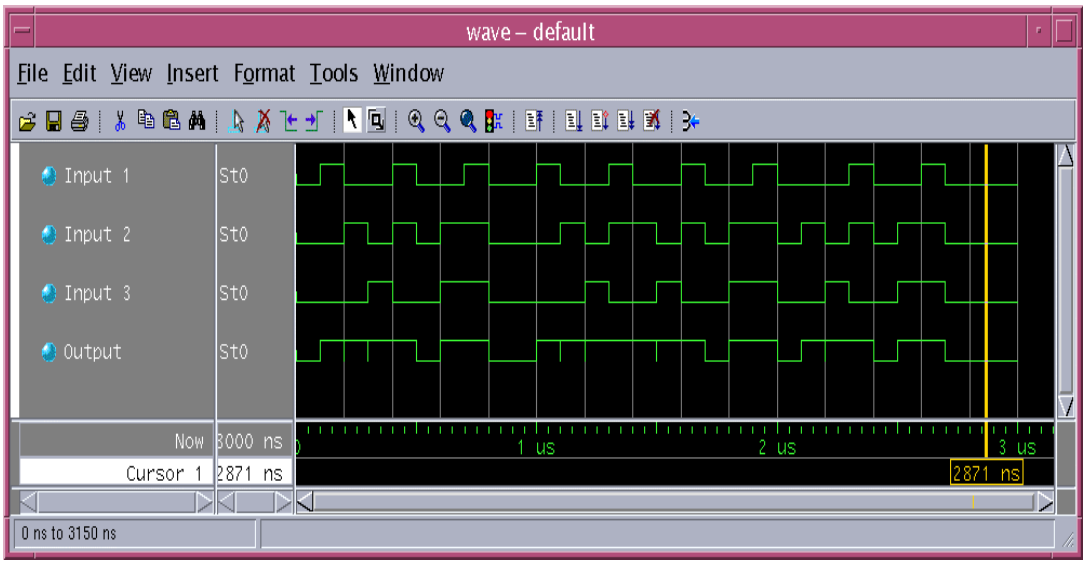

Figure 48 Simulation waveform of the Verilog code of the 3-input OR gate generated from VHDL by VHDL to Verilog converter

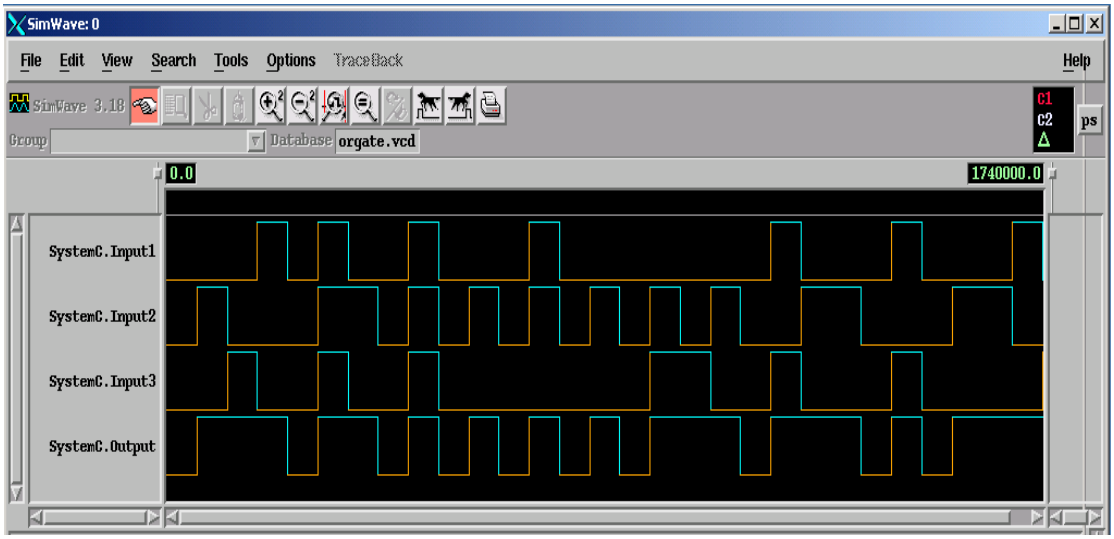

Figure 49 Simulation waveform of the SystemC code generated by Verilator of the 3-Input OR gate

5. Summary, Conclusion and Future Work

This research revolved around the plan to develop a method to make available the existing HDL IP in SystemC thereby reducing the simulation time of the large designs containing millions of gates becoming so important in this age of System-on-Chip. Research has shown the simulation time of SystemC code to be up to 3 times lesser than hand written VHDL and Verilog [24] [20].

In the first part of the research the conversion from HDL to SystemC was dealt with. A few tools that covert HDL to SystemC are available, some commercial and some free. The commercial tools could not be investigated due to non-availability. *Verilator* is a free tool, currently being developed, was used for converting Verilog code to SystemC. The tool converts most synthesizable Verilog into SystemC with little or no changes needed on the generated SystemC code. The flow from Verilog to SystemC and its verification is shown in Figure 50.

The major fallback of *Verilator* being that it currently does not produce platform independent code. The installation of *Verilator* on the system is required to be able to execute the SystemC code produced by it. This is because of the dependence of the generated SystemC files on the intermediate format System-Perl files. Also *Verilator* is available only for the Unix platform.

The source code for the *Verilator* tool is available and several advances can be made on the *Verilator* tool, the primary being able to generate SystemC code that

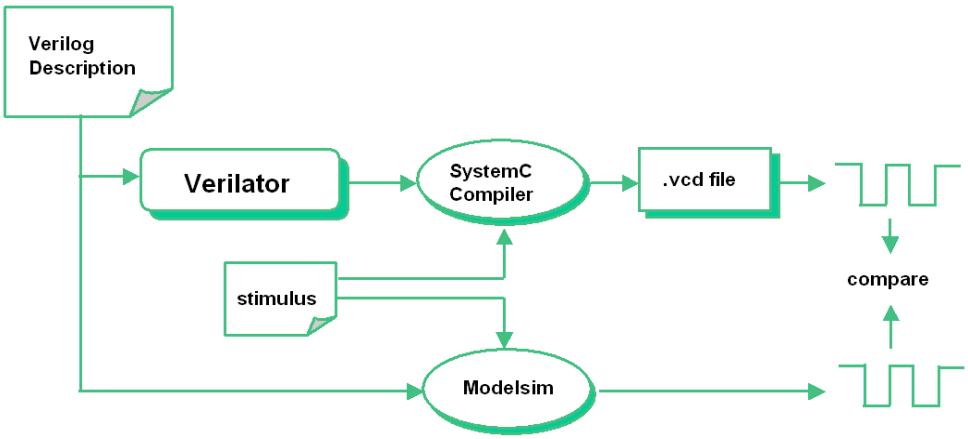

Figure 50 Verilog to SystemC flow

is platform independent. Future research could also focus on the development of *Verilator* for other platforms, especially Solaris. Improvement could also be made on the division and modulus operators that are currently limited to 32 bits.

In the second part of the research conversion among HDLs was dealt with. This is an area of research going on at several EDA companies with no solid results. The conversion of Verilog to VHDL or VHDL to Verilog is not always possible because of the different constructs supported by one language and not by the other.

Mentor Graphics' HDL Designer will convert VHDL to Verilog as long as there are no fancy user defined data types in the VHDL code. It will translate structural code i.e. block diagrams with no embedded blocks with a good accuracy. In case several modules are present in a design, each module will have to be converted individually. Verilog to VHDL translation can also be carried out using HDL Designer.

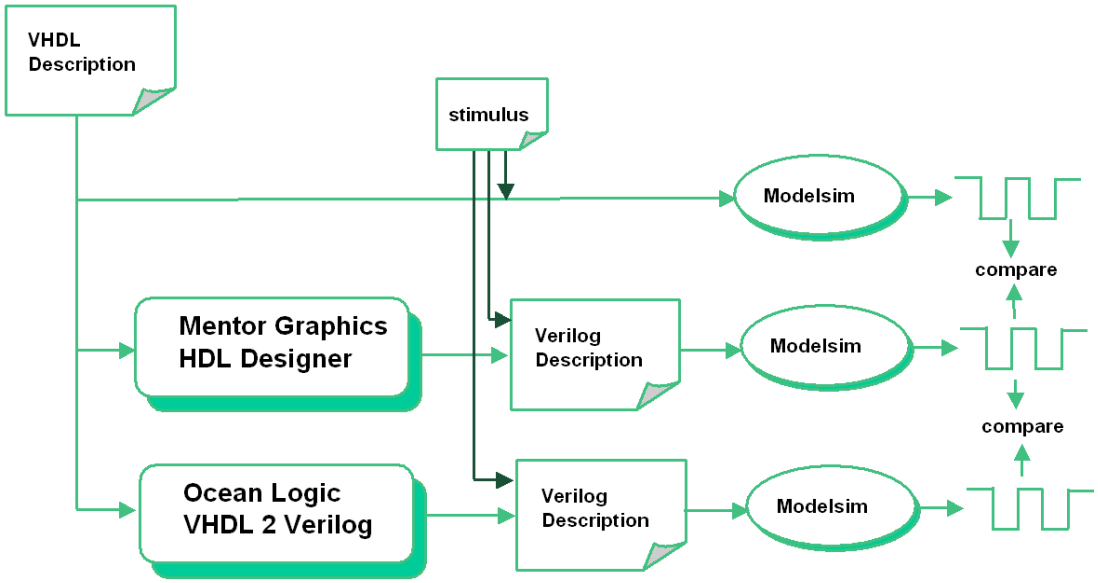

Figure 51 VHDL to Verilog flow

Ocean Logic's VHDL to Verilog converter will translate synthesizable VHDL RTL to Verilog. Here too when a design consists of a hierarchy, each individual file will have to be converted individually.

The Flow from VHDL to Verilog through various paths and its verification is is shown in Figure 51.

5.1 In a nut shell

This research intended to provide a working model at The University of Tennessee for the conversion of HDL to SystemC. The procedures for accomplishing the various conversion tasks have been well documented in Chapters 3 and 4. These will serve as a tutorial and also as a building block for further advancements by

making use of more options in the EDA tools that have not been discovered in this research. A summary of the cores converted using the various tools is shown in Table 1.

Table 1 Conversion summary

	Verilator	HDL Designer	Ocean Logic
CORDIC	OK	Not Applicable	Not Applicable
Single bit adder	OK	OK	OK
Four bit adder	OK	OK	OK
Multiplexer	OK	OK	OK
Flip Flop	OK	OK	OK
8-bit unsigned multiplier	Not Applicable	OK	OK
16-4 bit multiplexer	Not Applicable	OK	OK
RISC Processor	No	Not Applicable	Not Applicable
DES	Not Applicable	Not Attempted	No

References

[1] C. Noris Ip, Stuart Swan, “A Tutorial Introduction on the New SystemC Verification Standard,” 2003.

[2] Alex J. Wakefield, James Huggins, Robert Stets, “C++ as a Modeling and Verificatin Language.”

[3] Ando Ki, “Empirical Study of SystemC,” 2003.

[4] Cedric Alquier, Stephane Guerinneau, Lauro Rizzatti, Luc Burgan, “Co-Simulation Between SystemC and a New Generation Emulator,” 2003.

[5] G. Bollano, P. Garino, M. Turolla, M. Valemtini, “SystemC’s Impact on the Development of IP Libraries.”

[6] N. Agliada, A. Fin, F. Fummi, M. Martignano, G. Pravadelli, “On the Reuse of VHDL Modules into SystemC Designs.”

[7] L. Charest, E.M. Aboulhamid, “A VHDL / SystemC Comparison in handling Design Reuse.”

[8] G. Bollano, G.Cesana, S. Claretto, L. Licciardi, M. Paolini, M. Turolla, “The Virtual Intellectual Property Library: From Paradigm to Product.”

[9] Massimo Bombana, Francesco Bruschi, “SystemC VHDL co-simulation and synthesis in the HW domain.”

[10] Luc Semeria, Abhijit Ghosh, “Methodology for Hardware/Software Co-verification in C/C++.”

[11] Tim Kogel, Denis Bussaglia, “SystemC BasedDesign of an IP forwarding Chip with CoCentric SystemC Studio.”

[12] Ray Andraka, “A Survey of CORDIC algorithms for FPGA based computers.”

[13] Fredrick Doucet, Rajesh K. Gupta, “Microelectronic System-on-Chip Modelling using Objects and their Relationships.”

[14] Ney Calazans, Edson Moreno, Fabiano Hessel, Victor Rosa, Fernando Moraes, Everton Carara, “ From VHDL Register Transfer Level to SystemC Transaction Level Modelling: a Comparative Case Study.”

[15] Jon Connell, “Early hardware/Software integration using SystemC 2.0.”

[16] Alessandro Fin, Franco Fummi, Denis Signoretto, “ The Use of SystemC for Design Verification and Integration est of IP-Cores.”

[17] Sudeep Pasricha, “ Transaction level modeling of SoC with SystemC 2.0.”

[18] Guido Arnout, “C for System Level Design.”

[19] Stuart Swan, “An Introduction to System Level Modeling in SystemC 2.0.”

[20] Leila Mahmoudi Ayough, Ali Haj Abutalebi, Omid F. Nadjarbashi, Shaahin Hessabi, “ Verilog2SC: A Methodology for Converting Verilog HDL to SytemC.”

[21] Diederik Verkest, Joachim Kunkel, Frank Schirrmister, “System Level Design Using C++.”

[22] Wolfgand Mueller, “The Simulation Semantics of SystemC.”

[23] Guido Arnout, “SystemC standard.”

[24] Mario Steinert, Steffen Buch, David Slogsnat, “ Using SystemC for Hardware Design. Comparsion of results with VHDL, Cossap, CoCentric.”

[25] Brett Cline, “Why you or your replacement, will use SystemC for system design.”

[26] Grant Martin, “SystemC and the Future of Design Languages: Opportunities for users and Research.”

[27] J.R. Armstrong, Yuval Ronen, “Modeling with SystemC: A Case Study”.

[28] J. Bhasker, “A SystemC Primer,” Star Galaxy Publishing.

[29] Thorsten Grokter, Stan Liao, Grant Martin, Stuart Swan, “System Design with SystemC,” Kluwer Academic Publishers.

[30] Udaya Kamath, Rajita Kaundin, “System on Chip Designs,” June 2001.

[31] Synopsys Design Compiler User Guide

[32] Mentor Graphics HDL Designer User Guide

[33] www.veripool.com

[34] www.opencores.org

[35] www.synopsys.com

[36] www.mentor.com

[37] www.ocean-logic.com

[38] Advanced Intermediate Representation with Extensibility/Common Environment documentation.

Appendix

Appendix A Installation of Verilator

Verilator was built to support all Linux operating systems but certain changes in the installation files are necessary depending on the operating system. The downloaded files worked without any changes to them on RedHat Linux 8.0. *Verilator* is available for download from http://www.veripool.com/verilator.html. Installation of *Verilator* starts with the installation of the prerequisites for *Verilator*.

The prerequisites are

Perl 5.6.1 or later versions

Verilog-Perl

System-Perl

System C

C++ compiler, gunzip, make and tar are available by default with most Linux operating systems.

Installing Perl

Perl 5.6.1 or a later version is available for download from the following address: http://www.perl.com/language/info/software.html. It is always advisable to download and install the stable release version. The version of Perl used in this

project is Perl 5.8.0. The version of Perl available at the time of writing this is Perl 5.8.3. The following procedure was used to install Perl.

Create a directory in the home directory with name Perl

```
/sidd > mkdir perl
```

Download Perl from the above-mentioned web-address to this directory. This downloaded file will have the name stable.tar.gz. To unzip this file type

```
/sidd > cd perl
/sidd/perl > gunzip stable.tar.gz
```

This will unzip the zipped file and will produce a .tar file. To untar it

```
/sidd/perl/ > tar –xvf stable.tar
```

Untaring the files will produce another directory into which all the files are untared. Detailed instructions are provided in the file /perl-5.X.X/INSTALL to install Perl on a system resembling Unix. To install Perl type the following commands. Root privileges are required to install Perl.

```
/sidd/perl/perl-5.8.0 > rm -f config.sh Policy.sh
/sidd/perl/perl-5.8.0 > sh Configure –de
/sidd/perl/perl-5.8.0 > make
/sidd/perl/perl-5.8.0 > make test
/sidd/perl/perl-5.8.0 > make install
```

After the installation a message indicating the successful installation of Perl is displayed.

Installing Verilog-Perl

Verilog-Perl is available for download from the following web-address: http://www.veripool.com/verilog-perl.html. This package works on any system with Perl, G++ and Flex. The procedure followed to install Verilog-Perl is shown below.

Create a directory in the home area by the name vp.

```
/sidd > mkdir vp
```

Download Verilog-Perl from the above-mentioned web-address to this directory. Unzip and untar the downloaded file.

```
/sidd/vp > cd vp
/sidd/vp > gunzip Verilog-Perl-2.226.tar.gz
/sidd/vp > tar –xvf Verilog-Perl-2.226.tar
```

This will create a directory by the name Verilog-Perl-2.226 in the vp directory. Type

```
/sidd/vp > cd Verilog-Perl-2.266
/sidd/vp/Verilog-Perl-2.226 > perl Makefile.PL
```

to configure Verilog-Perl for your system. To compile Verilog-Perl type

```
/sidd/vp/Verilog-Perl-2.226 > make
```

To test the installation and to install all programs, data files and documentation type

```
/sidd/vp/Verilog-Perl-2.226 > make test
/sidd/vp/Verilog-Perl-2.226 > make install
```

A message indicating the completion of all tests indicating the successful installation of Verilog-Perl on the computer is displayed.

Installing System-Perl

System-Perl is available for download from the following web-address: http://www.veripool.com/systemperl.html. This package should work on any system with Perl, G++, Bison and Flex. The procedure followed to install System-Perl is shown below.

Create a directory in the home area by the name sp.

```
/sidd > mkdir sp
```

Download System-Perl from the above-mentioned web-address to this directory. Unzip and untar the downloaded file.

```
/sidd > cd sp
/sidd/sp > gunzip SystemPerl-1.145.tar.gz
/sidd/sp > tar –xvf SystemPerl-1.145.tar
```

This will create a directory by the name SystemPerl-1.145 in the sp directory. Type

```
/sidd/sp/ > cd SystemPerl-1.145
/sidd/sp/SystemPerl-1.145 > perl Makefile.PL
```

to configure System-Perl for your machine. Type

```
/sidd/sp/SystemPerl-1.145 > make
```

to compile System-Perl and to test the installation type

```
/sidd/sp/SystemPerl-1.145 > make test
```

To install the documentation and the data files type

```
/sidd/sp/SystemPerl-1.145 > make install
```

A message indicating the completion of all tests indicating the successful installation of System-Perl on the computer is displayed.

Installing SystemC

SystemC is available for download from http://www.systemc.org. This project was carried out on SystemC version 2.0.1. The procedure followed to install SystemC is shown below.

Create a directory with name sc in the home area and download the SystemC installation file from the above mentioned website.

```
/sidd > mkdir sc
```

Move into this directory and unzip and untar the downloaded SystemC installation file.

```
/sidd > cd sc
/sidd/sc > gunzip systemc-2.0.1.tar.gz
/sidd/sc > tar –xvf systemc-2.0.1.tar
```

This will untar all the installation files into a directory named systemc-2.0.1 Move into this directory and create a temporary directory

```
/sidd/sc > cd systemc-2.0.1
/sidd/sc/systemc-2.0.1 > mkdir objdir
```

Move into this directory and set the following environment variables

```
/sidd/sc/systemc-2.0.1 > cd objdir
/sidd/sc/systemc-2.0.1/objdir > export CXX=g++
```

Configure the package for your system

```
/sidd/sc/systemc-2.0.1/objdir > ../configure
```

While the 'configure' script is running, which takes a few moments, it prints messages to inform you of the features it is checking. It also detects the platform.

Compile the package by typing

```
/sidd/sc/systemc-2.0.1/objdir > gmake
```

To install the package type

```
/sidd/sc/systemc-2.0.1/objdir > gmake install
```

The temporary directory is no longer needed and it may be removed

```
/sidd/sc/systemc-2.0.1/objdir > cd ..
/sidd/sc/systemc-2.0.1/ > rm –rf objdir
```

To check for the proper installation of SystemC examples provided along with the SystemC package may be executed.

The examples are located in the /systemc-2.0.1/examples directory, Move to this directory and to one of the examples

```
/sidd/sc/systemc-2.0.1/examples/ > cd pipe
```

Make and execute this example

```
../systemc-2.0.1/examples/pipe> make–f  Makefile.gcc
```

An executable file is created with name run.x. Execute this file

```
/sidd/sc/systemc-2.0.1/examples/pipe > ./run.x
```

The successful execution of this indicates the successful installation of SystemC.

Installing Verilator

Verilator was installed after the installation of all the prerequisites. It is available for download from http://www.veripool.com/verilator.html. This project was carried out with *Verilator-3.110*. The latest version available at time of writing this is version 3.210. The procedure followed to install *Verilator* is shown below.

Create a directory in the home area by name ‘ver’. Download *Verilator* from the above mentioned web-address to this directory and unzip and untar the downloaded file.

```
/sidd/ver > gunzip verilator-3.110.tar.gz
/sidd/ver/ > tar –xvf verilator-3.110.tar
```

This will create a directory by name verilator-3.110 and all the files are untared into this directory. Move to this directory and configure *Verilator* for the system.

```
/sidd/ver > cd verilator-3.110
/sidd/ver/verilator-3.110 > ./configure
```

Type make to compile *Verilator*

```
/sidd/ver/verilator-3.110 > make
```

To test the whole package of verilator type make test

```
/sidd/ver/verilator-3.110 > make test
```

To test the SystemC converter alone type make test_sp

```
/sidd/ver/verilator-3.110 > make test_sp
```

It is more important for this test to successfully work than the comprehensive test. The make test_sp command converts the Verilog files present in the test_v directory to SystemC files in the /test_sp/objdir directory

Vita

Mr. Siddhartha Devalapalli was born (1978) and brought up in Hyderabad, India. He did his schooling from The Hyderabad Public School, Hyderabad, India and joined The Little Flower Junior College, Hyderabad in 1994. From there he went on to pursue a professional career in engineering at The Bangalore University, Bangalore and graduated with a Bachelors in Electronics and Communications Engineering in 2000. He joined University of Tennessee, Knoxville in August 2001 to pursue Masters of Science in Electrical Engineering. He has worked as a Graduate Teaching Assistant at The Department of Electrical and Computer Engineering at The University of Tennessee since January 2002 till present. He plans to graduate with a Masters degree in Electrical Engineering in August 2004.