System-on-Chip Environment

SCE Version 2.2.0 Beta

Tutorial

Samar Abdi
Junyu Peng
Haobo Yu
Dongwan Shin
Andreas Gerstlauer
Rainer Doemer
Daniel Gajski

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425
+1 (949) 824-8919
http://www.cecs.uci.edu

System-on-Chip Environment: SCE Version 2.2.0 Beta; Tutorial

by Samar Abdi, Junyu Peng, Haobo Yu, Dongwan Shin, Andreas Gerstlauer, Rainer Doemer, and Daniel Gajski

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425
+1 (949) 824-8919
http://www.cecs.uci.edu

Published July 23, 2003

Table of Contents

Chapter 1. Introduction

The basic purpose of this tutorial is to guide a user through our System-on-Chip design environment (SCE). SCE helps designers to take an abstract functional description of the design and produce an implementation. We begin with a brief overview of our SoC methodology by describing the design flow and various abstraction levels. The overview also covers the user interfaces and the tools that support the design flow.

We then describe the example that we use throughout this tutorial. We selected the GSM Vocoder as an example for a variety of reasons. For one, the Vocoder is a fairly large design and is an apt representative of a typical component of a System-on-Chip design. Moreover, the functional specification of the Vocoder is well defined and publicly available from the European Telecommunication Standards Institute (ETSI).

The tutorial gives a step by step illustration of using the System-on-Chip Environment. Screenshots of the GUI are presented to aid the user in using the various features of SCE. (Please note that, depending on your specific version of the System-on-Chip Environment SCE and your system settings, the screen shots shown in this document may be slightly different from the actual display on your screen.) Over the course of this chapter, the user is guided on synthesizing the Vocoder model from an abstract specification to a clock cycle accurate implementation. The screenshots at each design step are supplemented with brief observations and the rationale for making design decisions. This would help the designer to gain an insight into the design process instead of merely following the steps. We wind up the tutorial with a conclusion and references. This tutorial assumes that the readers of this tutorial have basic knowledge of system design tasks and flow. In case the reader feels difficulty going following this tutorial, he can always go to the Appendix A: FAQ (Frequently Asked Questions) at the end of the tutorial to seek more explanation.

1.1. Motivation

System-on-Chip capability introduces new challenges in the design process. For one, co-design becomes a crucial issue. Software and Hardware must be developed together. However, both Software and Hardware designers have different views of the system and they use different design and modeling techniques.

Secondly, the process of system design from specification to mask is long and elaborate. The process must therefore be split into several steps. At each design step, models must be written and relevant properties must be verified.

Thirdly, the system designers are not particularly fond of having to learn different languages. Moreover, writing different models and validating them for each step in the design process is a huge overkill. Designers prefer to create solutions rather than write several models to verify their designs.

It is with these aspects and challenges in mind that we have come up with a System-on-Chip Environment that takes off the drudgery of manual repetitive work from the designers by generating each successive model automatically according to the decisions made by the designers.

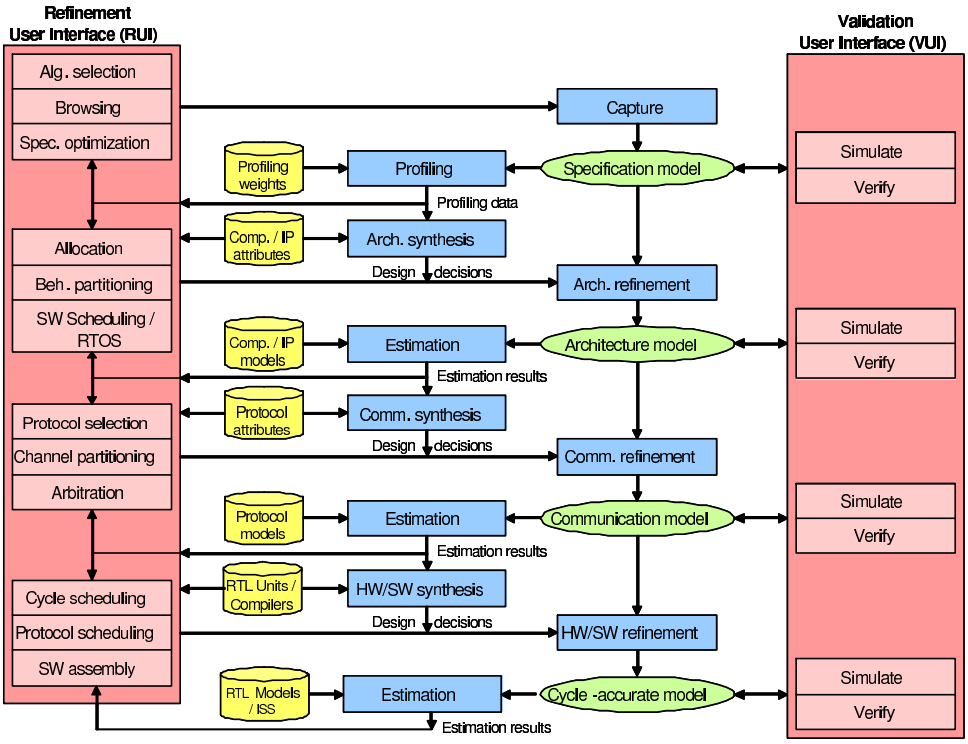
1.2. SCE Goals

SCE represents a new technology that allows designers to capture system specification as a composition of C-functions. These are automatically refined into different models required at each step of the design process. Therefore designers can devote more effort to the creative part of designing and the tools can create models for validation and synthesis. The end result is that the designers do not need to learn new system level design languages (SystemC, SpecC, Superlog, etc.) or even the existing Hardware Description Languages (Verilog, VHDL).

Consequently, the designers have to enter only the golden specification of the design and make design decisions interactively in SCE. The models for simulation, synthesis and verification are generated automatically.

1.3. Models for System Design

Figure 1-1. System-on-Chip Environment



The System-on-Chip design environment is shown in Figure 1-1. It consists of 4 levels of model abstraction, namely specification, architecture, communication and cycle-accurate models. Consequently, there are 3 refinement steps, namely architecture refinement, communication refinement and HW/SW refinement. These refinement steps are preformed in the top-down order as shown. As shown in Figure 1-1, we begin with an abstract specification model. The specification model is untimed and has only the functional description of the design. Architecture refinement transforms this specification to an architecture model. It involves partitioning the design and mapping the partitions onto the selected components. The architecture model thus reflects the intended architecture for the design. The next step, communication refinement, adds system busses to the design and maps the abstract communication between components onto the busses. The resulted design is a timing accurate communication model (bus functional model). The final step is HW/SW refinement which produces clock cycle accurate RTL model for

the hardware components and instruction set specific assembly code for the processors. All models have well defined semantics, are executable and can be validated through simulation.

1.4. System-on-Chip Environment

The SCE provides an environment for modeling, synthesis and validation. It includes a graphical user interface (GUI) and a set of tools to facilitate the design flow and perform the aforementioned refinement steps. The two major components of the GUI are the Refinement User Interface (RUI) on the left and the Validation User Interface (VUI) on the right as shown in Figure 1-1. The RUI allows designers to make and input design decisions, such as component allocation, specification mapping. With design decisions made, refinement tools can be invoked inside RUI to refine models. The VUI allows the simulation of all models to validate the design at each stage of the design flow.

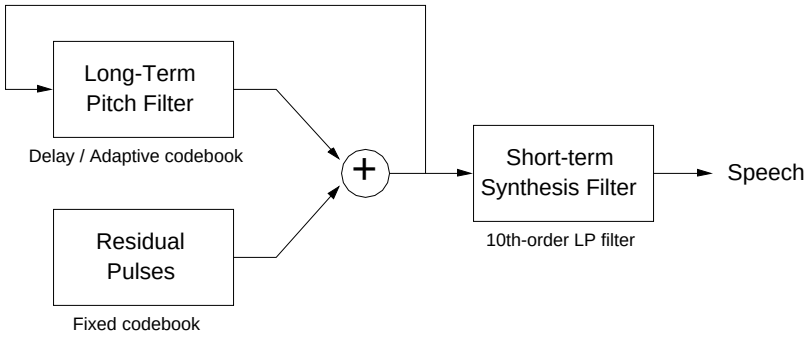
Each of the boxes corresponds to a tool which performs a specific task automatically. A profiling tool is used to obtain the characteristics of the initial specification, which serves as the basis for architecture exploration. The refinement tool set automatically transforms models based on relevant design decisions. The estimation tool set produces quality metrics for each intermediate models, which can be evaluated by designers.

With the assistance of the GUI and tool set, it is relatively easy for designer to step through the design process. With the editing, browsing and algorithm selection capability provided by RUI, a specification model can be efficiently captured by designers. Based on the information profiled on the specification, designers input architectural decisions and apply the architecture refinement tool to derive the architecture model. If the estimated metrics are satisfactory, designers can focus on communication issues, such as protocol selection and channel partitioning. With communication decisions made, the communication refinement tool is used to generate the communication model. Finally, the implementation model is produced in the similar fashion. The implementation model is ready for RTL synthesis.

We are currently in the process of developing tools for automating the synthesis tasks for system level design shown in the exploration engine. The tutorial presents automatic RTL synthesis. The next challenge is to automatically perform architecture and communication synthesis.

1.5. Design Example: GSM Vocoder

Figure 1-2. GSM Vocoder



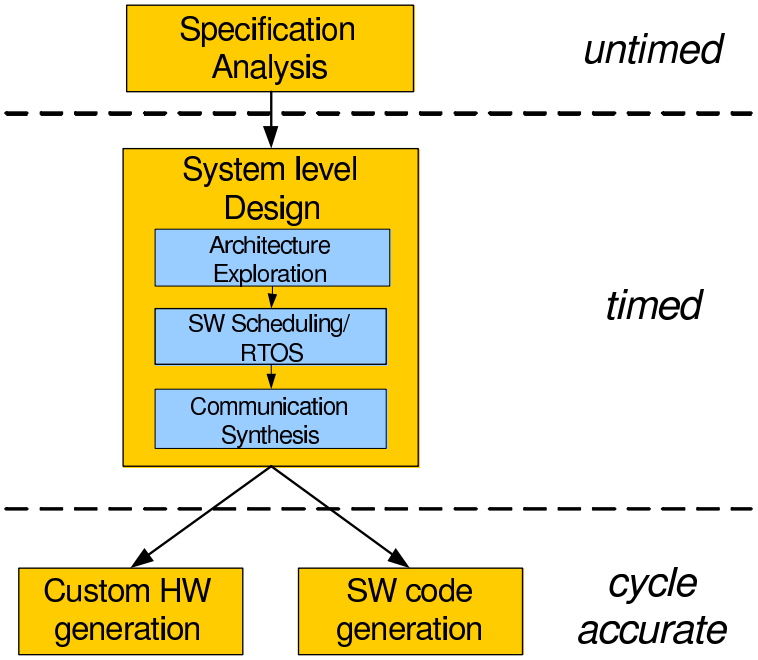
The example design used throughout this tutorial is the GSM Vocoder system , which is employed worldwide for cellular phone networks. Figure 1-2 shows the GSM Vocoder speech synthesis model. A sequence of pulses is combined with the output of a long term pitch filter. Together they model the buzz produced by the glottis and they build the excitation for the final speech synthesis filter, which in turn models the throat and the mouth as a system of lossless tubes.

The example used in this tutorial encodes speech data comprised of frames. Each frame in turn comprises of 4 sub-frames. Overall, each sub-frame has 40 samples which translate to 5 ms of speech. Thus each frame has 20 ms of speech and 160 samples. Each frame uses 244 bits. The transcoding constraint (ie. back to back encoder/decoder) is less than 10 ms for the first sub-frame and less than 20 ms for the whole frame (consisting of 4 sub-frames).

The vocoder standard, published by the European Telecommunication Standards Institute (ETSI), contains a bit-exact reference implementation of the standard in ANSI C. This reference code was taken as the the basis for developing the specification model. At the lowest level, the algorithms in C could be directly reused in the leaf behaviors without modification. Then the C function hierarchy was converted into a clean and efficient hierarchical specification by analyzing dependencies, exposing available parallelism, etc. The final specification model is composed of 9139 lines of SpecC code, which contains 73 leaf behaviors.

1.6. Organization of the Tutorial

Figure 1-3. Task flow for system design with SCE



The tasks in system design with SCE are organized as shown in Figure 1-3. Each of the tasks is explained in a separate chapter in this tutorial. We will start with a specification model and show how to get started with SCE. At this level, we will be working with untimed functional models. Following that, we will look at system level exploration and refinements, where the involved models will have a quantitative notion of time. Once we get a system model with well defined HW and SW components and the interfaces between them, we will proceed to generate custom hardware and processor specific software. These final steps will produce cycle accurate models.

Each design task is composed of several steps like model analysis, browsing, generation of new models and simulation. Not all these steps are crucial for the demo to proceed smoothly. Some steps are marked as optional and may be avoided during the course of this tutorial. If the designer is sufficiently comfortable with the tool's result, he or she can avoid the typically optional steps of simulation and code viewing.

If the designer is booting from the CD-ROM, the setup is already prepared.

Otherwise, the designer may follow the following steps to set up the demo. Start with a new shell of your choice. If you are working with a c-shell, run "source $SCE_INSTALLATION_PATH/bin/setup.csh". If you are working with bourne shell, run "$SCE_INSTALLATION_PATH/bin/setup.sh". Now run "setup_demo" to setup the demonstration in the current directory. This will add some new files to be used during the demo.

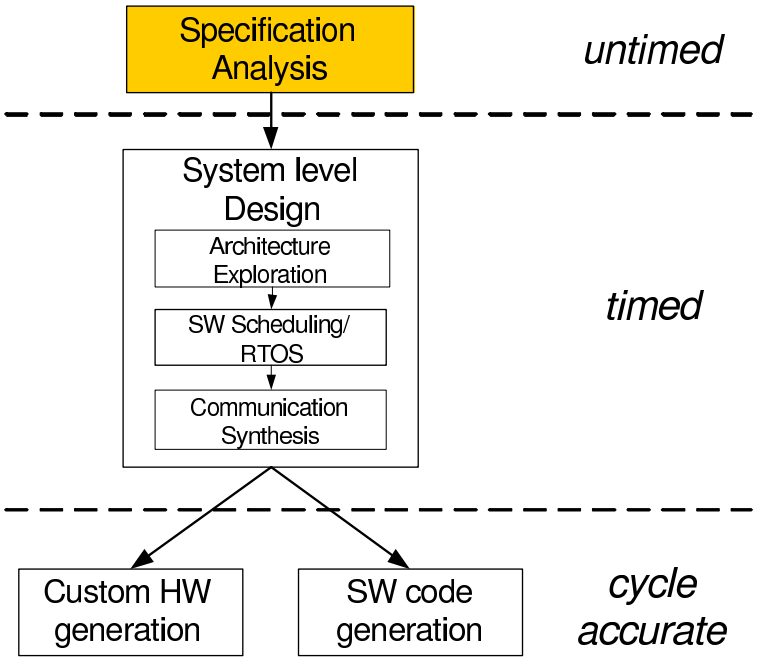
Acknowledgment:

The authors would like to thank Tsuneo Kinoshita of NASDA, Japan for his patience in going through the tutorial and helping us make it more understandable and comprehensive. We would also like to thank Yoshihisa Kojima of the University of Tokyo for his help in uncovering several mistakes in the tutorial's text.

Chapter 2. System Specification Analysis

2.1. Overview

Figure 2-1. Specification analysis using SCE



The system design process starts with the specification model written by the user to specify the desired system functionality. It forms the input to the series of exploration and refinement steps in the SoC design methodology. Moreover, the specification model defines the granularity for exploration through the size of the leaf behaviors. It exposes all available parallelism and uses hierarchy to group related functionality and manage complexity.

In this chapter, we go through the steps of creating a project in SCE and initiating the system design process as highlighted in Figure 2-1. The various aspects of the specification are observed through simulation and profiling. Also, the model is graphically viewed with the help of SCE tools.

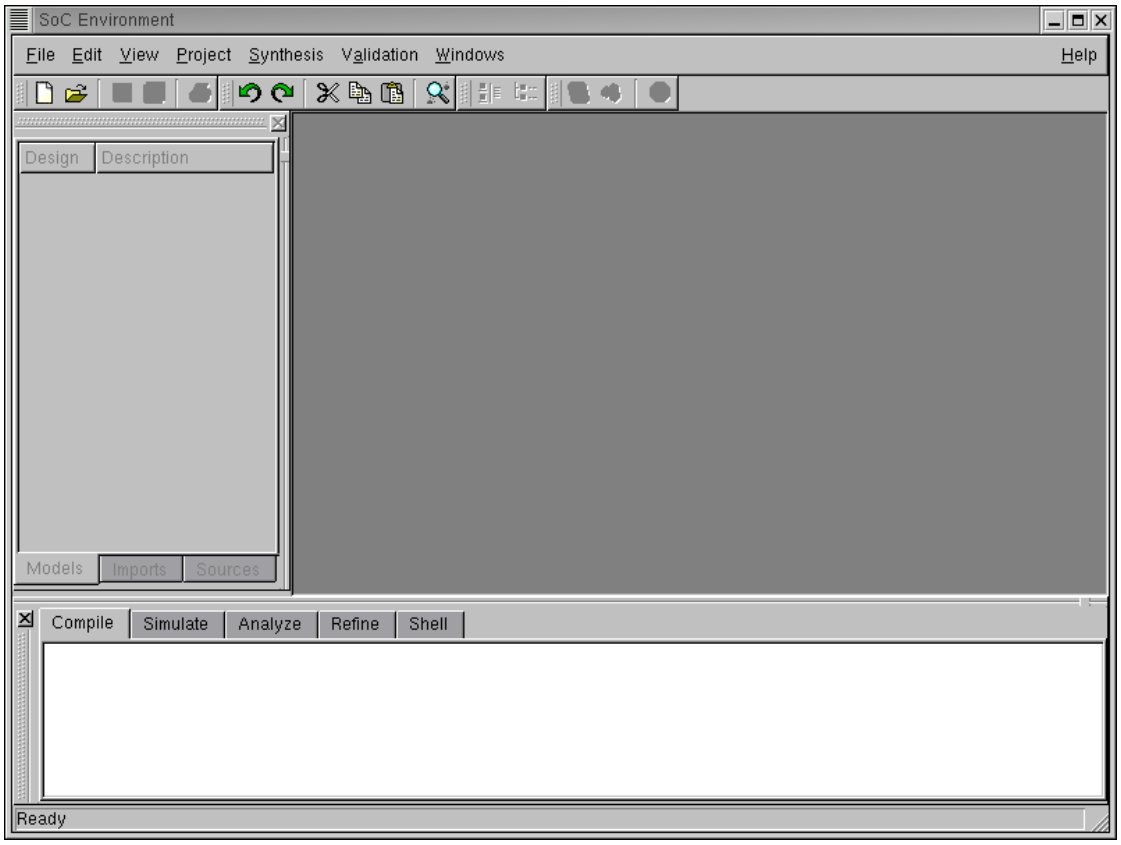
2.2. Specification Capture

The system design process starts with the specification model written by the user to specify the desired system functionality. It forms the input to the series of exploration and refinement steps in the SoC design methodology. Moreover, the specification model defines the granularity for exploration through the size of the leaf behaviors. It exposes all available parallelism and uses hierarchy to group related functionality and manage complexity.

In this section, we go through the steps of creating a project in SCE and initiating the system design process. The various aspects of the specification are observed through simulation and profiling. Also, the model is graphically viewed with the help of SCE tools.

The models that we will deal with in this phase of system design are untimed functional models. The tasks of the system specification, referred to as behaviors in our parlance, follow a causal order of execution. The main idea in this section is to introduce the user to the SCE GUI and to demonstrate the capability of graphically viewing the behaviors and their organization in the specification model.

2.2.1. SCE window

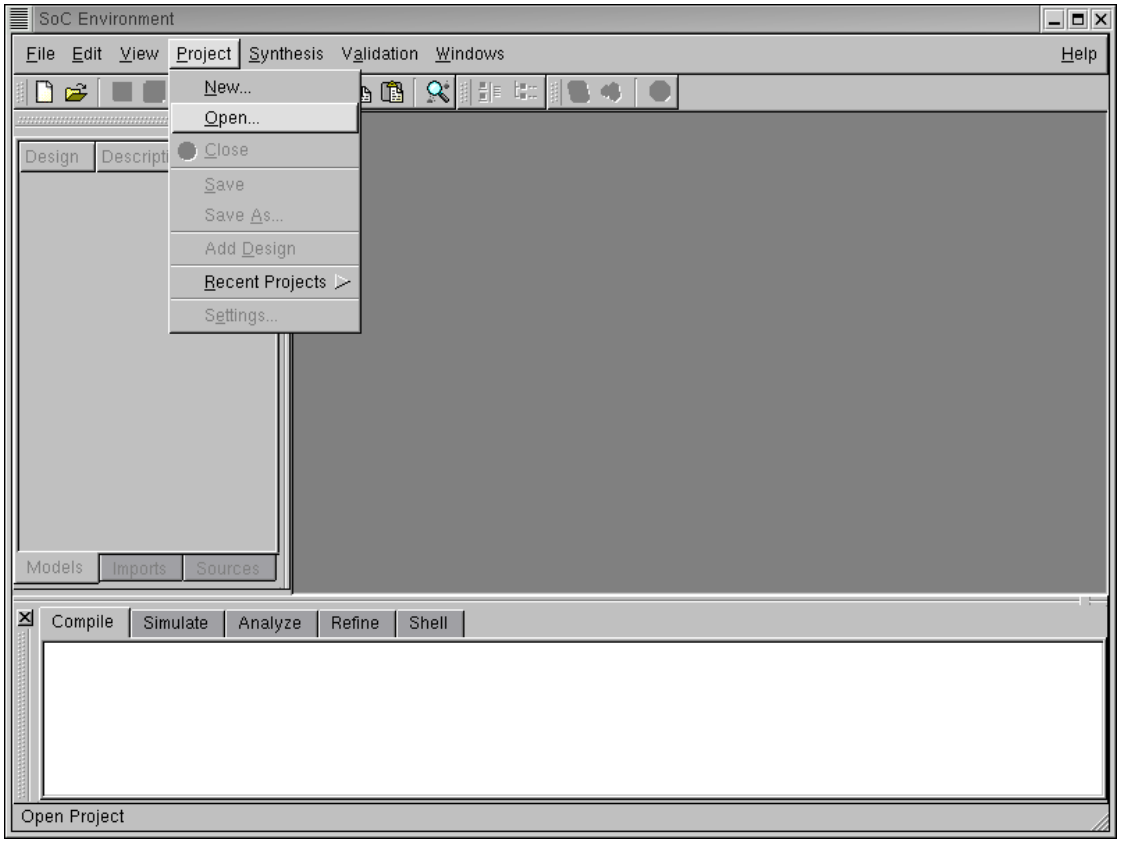


To launch the SCE GUI, simply run "sce" from the shell prompt. On launching the System-on-Chip Environment (SCE), we see the above GUI. The GUI is divided broadly into three parts. First is the "project management" window on the top left part of the GUI, which maintains the set of models in the open projects. This window becomes active once a project is opened and a design is added to it. Secondly, we have the "design management" window on the top right where the currently active design is maintained. It shows the hierarchy tree for the design and maintains various statistics associated with it. Finally, we have the "logging" window at the bottom of the GUI, which keeps the log of various tools that are run during the course of the demo. We keep logs of compilation, simulation, analysis and refinement of models.

The GUI also consists of a tool bar and shortcuts for menu items. The File menu handles file related services like opening designs, importing models etc. The Edit menu is for editing purposes. The View menu allows various methods of graphically viewing the design. The Project menu manages various projects. The Synthesis menu provides for launching the various refinement tools and making synthesis decisions. The Validation

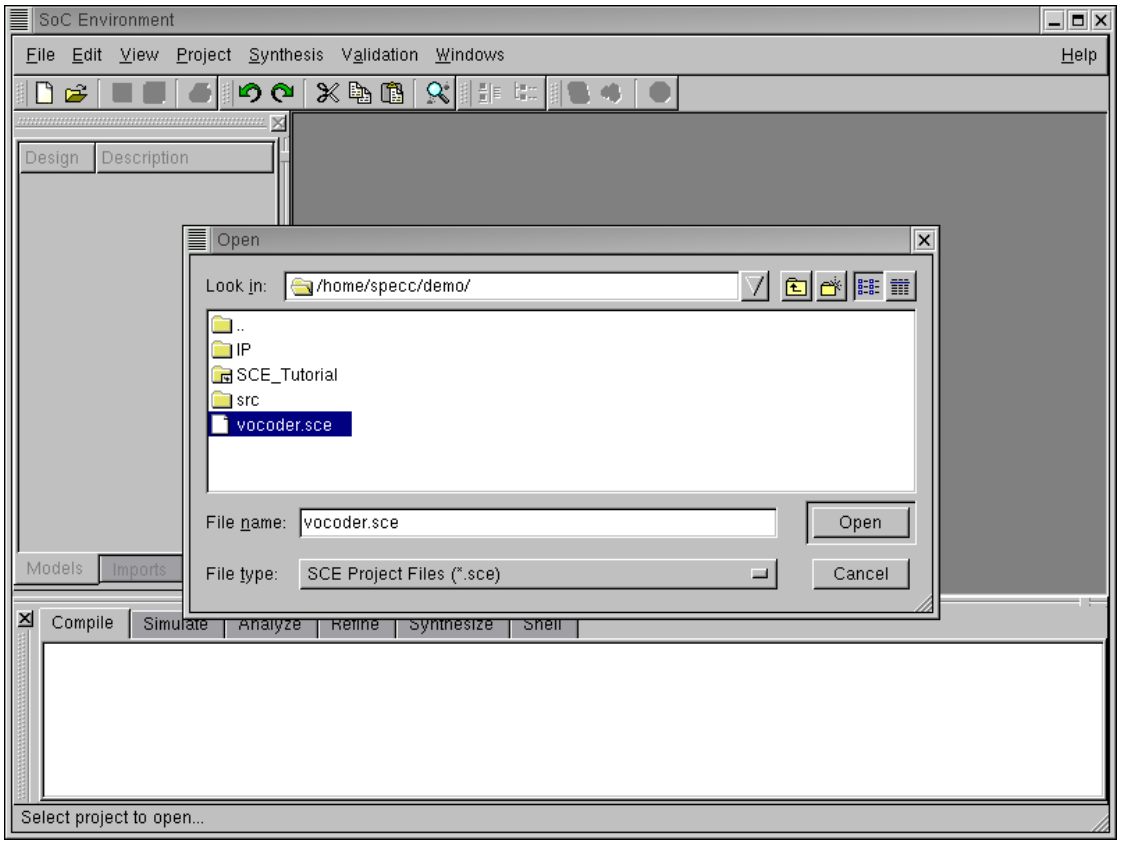
menu is primarily for compiling or simulating models.

2.2.2. Open project



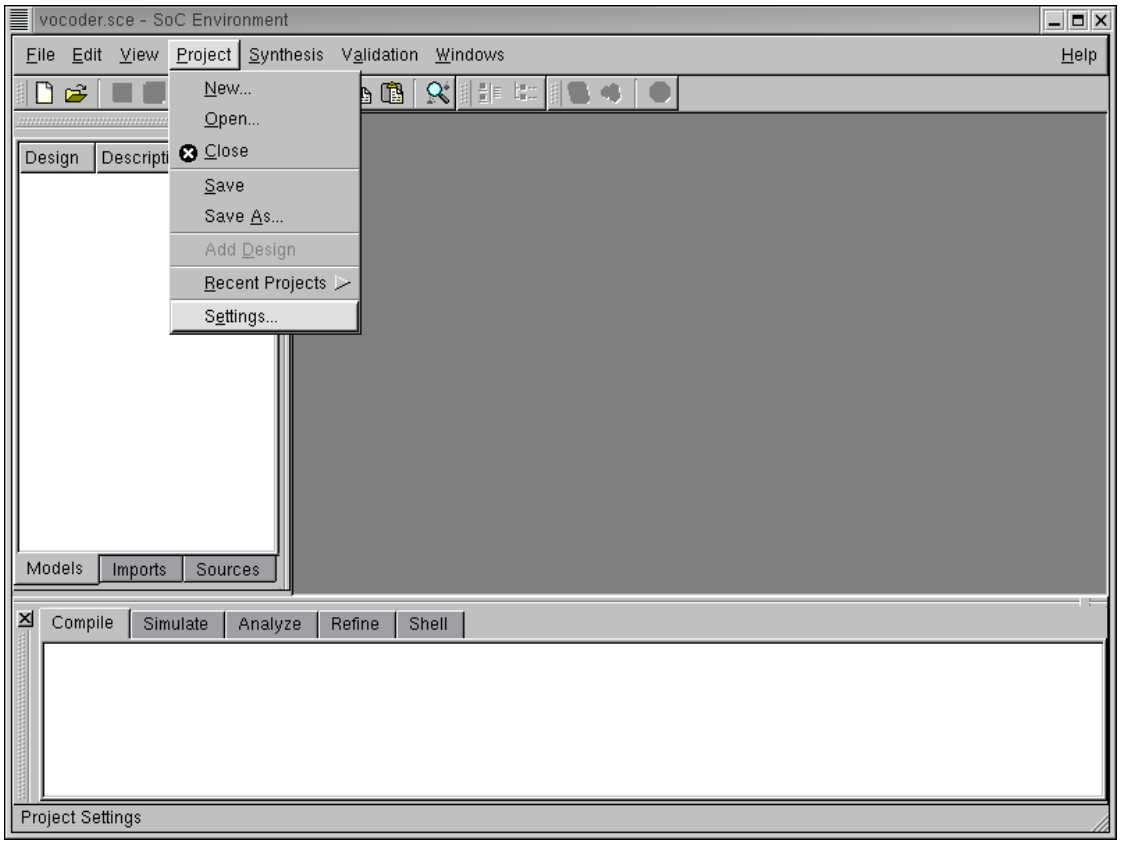
The first step in working with SCE is opening a project. A project is associated with every design process since each design might impose a different set of databases or dependencies. The project is hence used by the designer to customize the environment for a particular design process. We begin by selecting Project⟶Open from the menu bar.

2.2.2.1. Open project (cont'd)



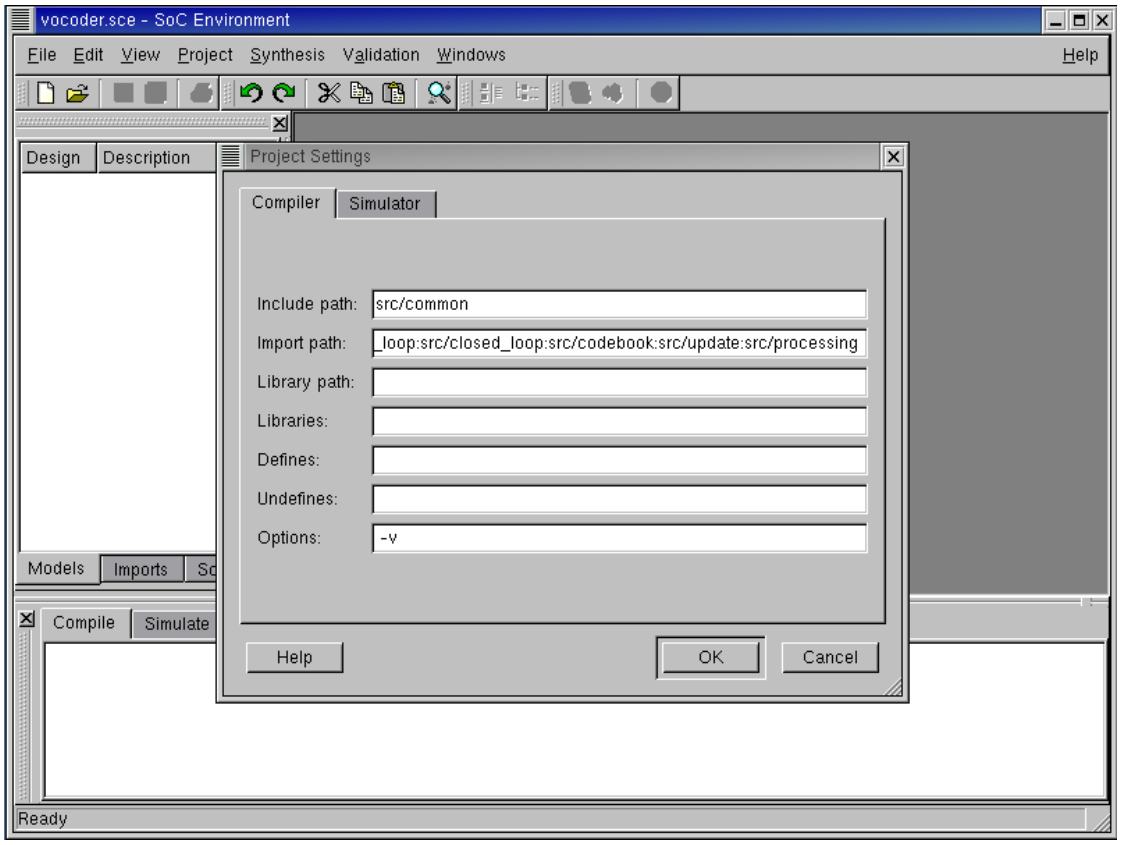
A Open file window pops up. For the purpose of the demo, a project is pre-created. We simply open it by selecting the project "vocoder.sce" and left click on Open button on the right corner of the the pop-up window.

2.2.2.2. Open project (cont'd)



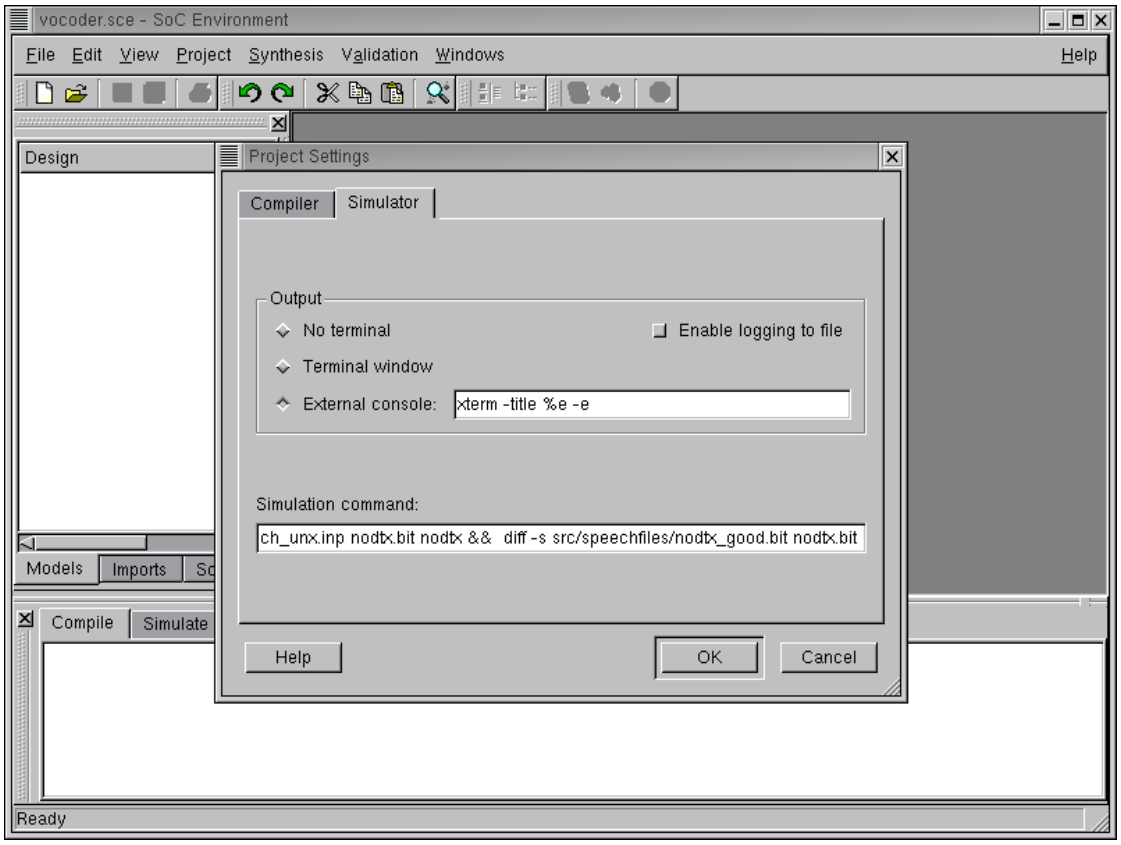
Since we need to ensure that the paths to dependencies are correctly set, we now check the settings for this precreated "vocoder.sce" project by selecting Project⟶Settings... from the top menu bar.

2.2.2.3. Open project (cont'd)



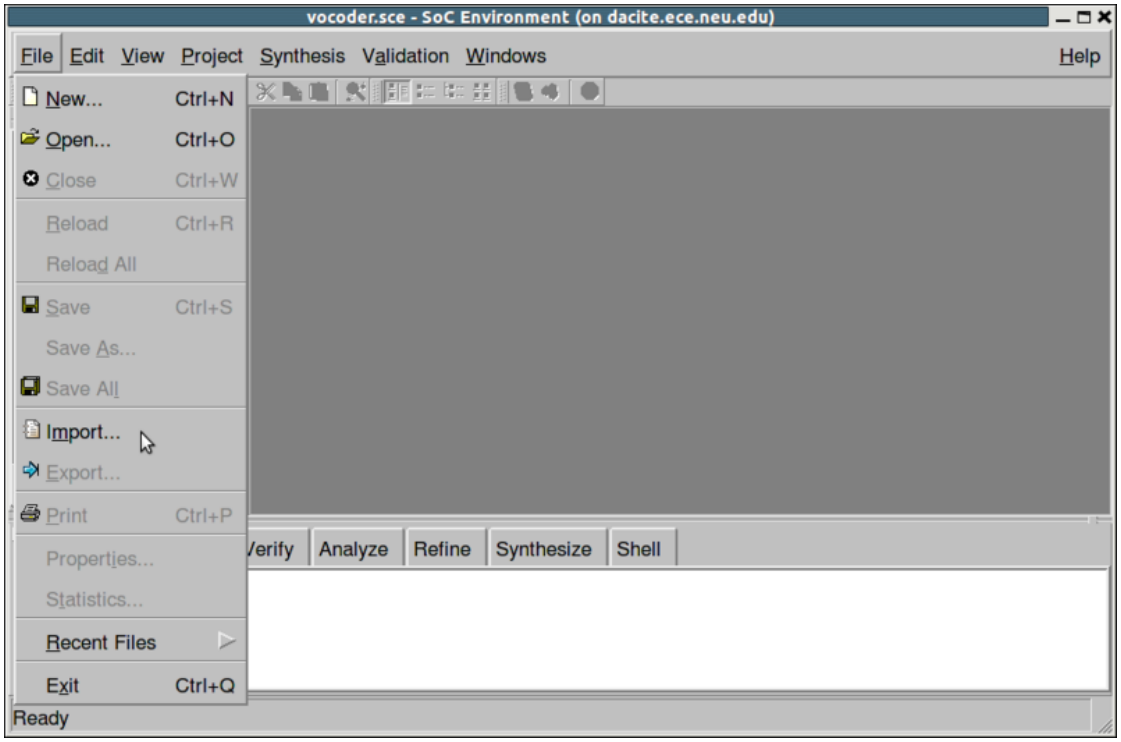
We now see the compiler settings showing the import path for the model's libraries and the '-v' (verbose) option. The Include path setting gives the path which is searched for header files. The Import path is searched for files imported into the model. The Library path is used for looking up the libraries used during compilation. There are also settings provided for specifying which libraries to link against, which macros to define and which to undefine. These settings basically form the compilation command. To check the simulator settings, left click on the Simulator tab.

2.2.2.4. Open project (cont'd)



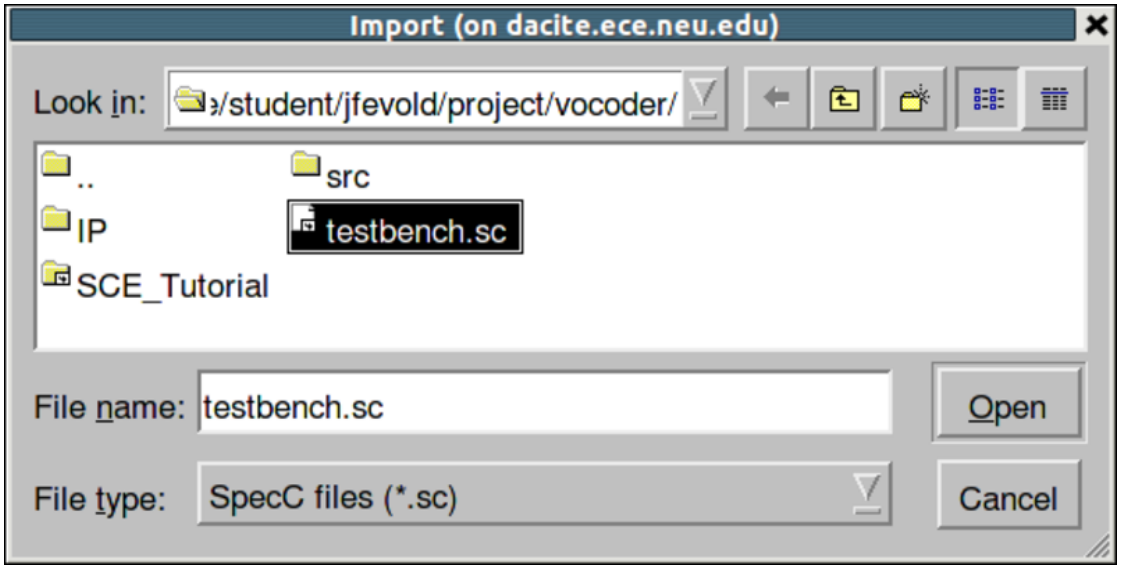
We now see the simulator settings showing the simulation command for the "vocoder.sce" project. There are settings available to direct the output of the model simulation. As can be seen, the simulation output may be directed to a terminal, logged to a file or dumped to an external console. For the demo, we direct the output of the simulation to an xterm. Also note that the simulation command may be specified in the settings. This command is invoked when the model is validated after compilation. The vocoder simulation processes 163 frames of speech and the output is matched against a golden file. Press OK to proceed.

2.2.3. Open specification model



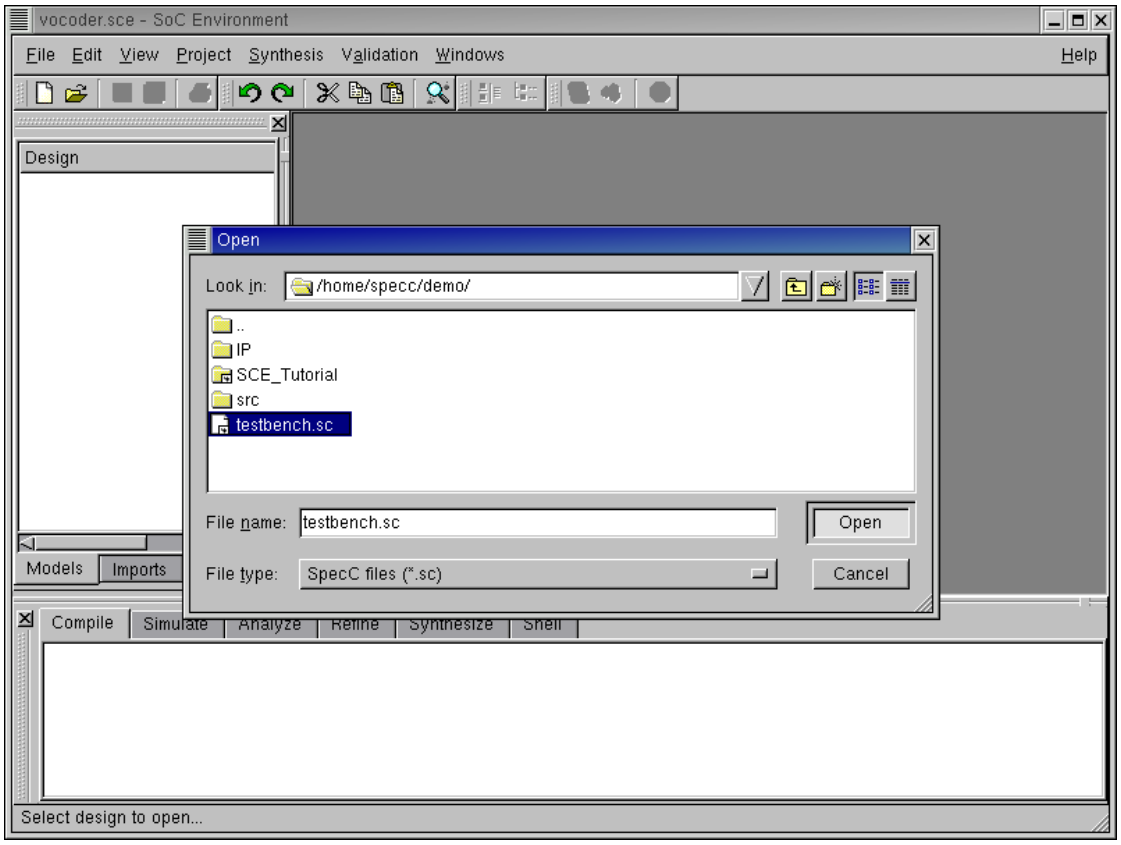
We start with the specification that was already captured as a model. We open this model to see if it meets the desired behavior. Once the model is validated to be "golden", we will start refining it and adding implementation details to it. We open the specification model for the Vocoder example by selecting File⟶Import from the menu bar.

2.2.3.1. Open specification model (cont'd)



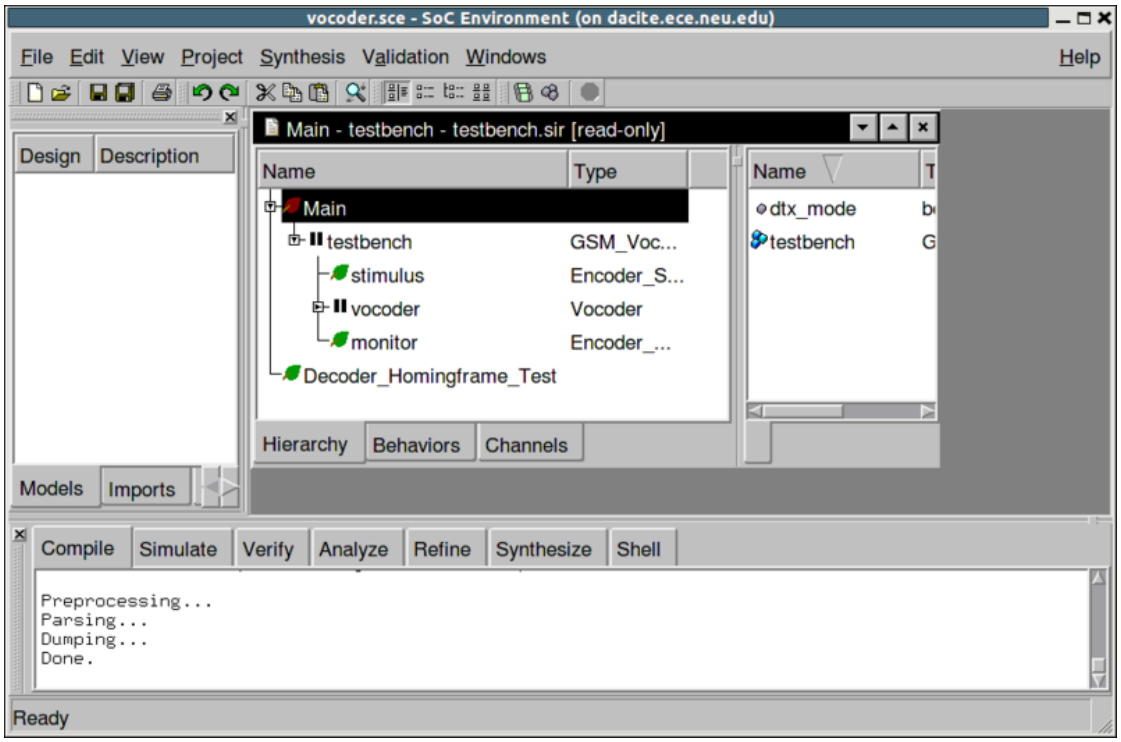
A file Open window pops up showing the SpecC internal representation (SIR) files. The internal representation files are a collection of data structures used by the tools in the environment. They uniquely identify a SpecC model. At this time however, the design is available only in its source form. We therefore need to start with the sources. Select "SpecC files (*.sc)" to view the source files.

2.2.3.2. Open specification model (cont'd)



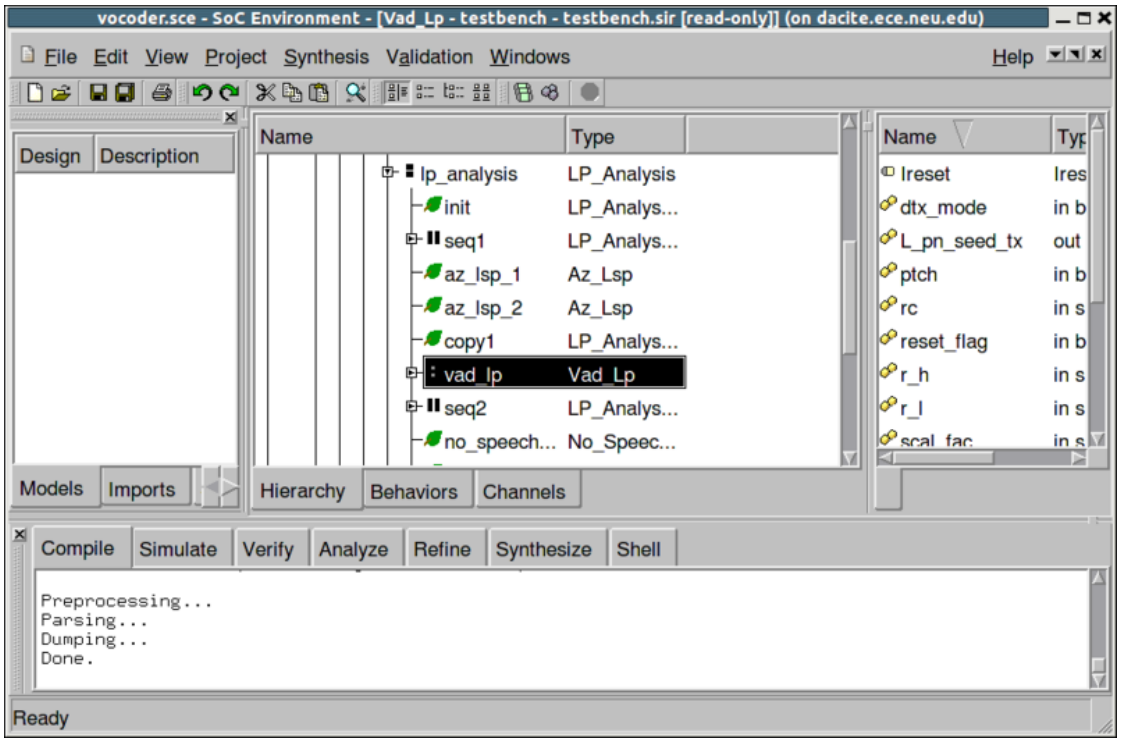
The Open is updated to show the available source files of the GSM Vocoder design specification. Select the file containing the top hierarchy of the model. In this case, the file is "testbench.sc". The testbench instantiates the design-under-test (DUT) and the corresponding modules for triggering the test vectors and for observing the outputs. To open this file Left click on Open.

2.2.3.3. Open specification model (cont'd)



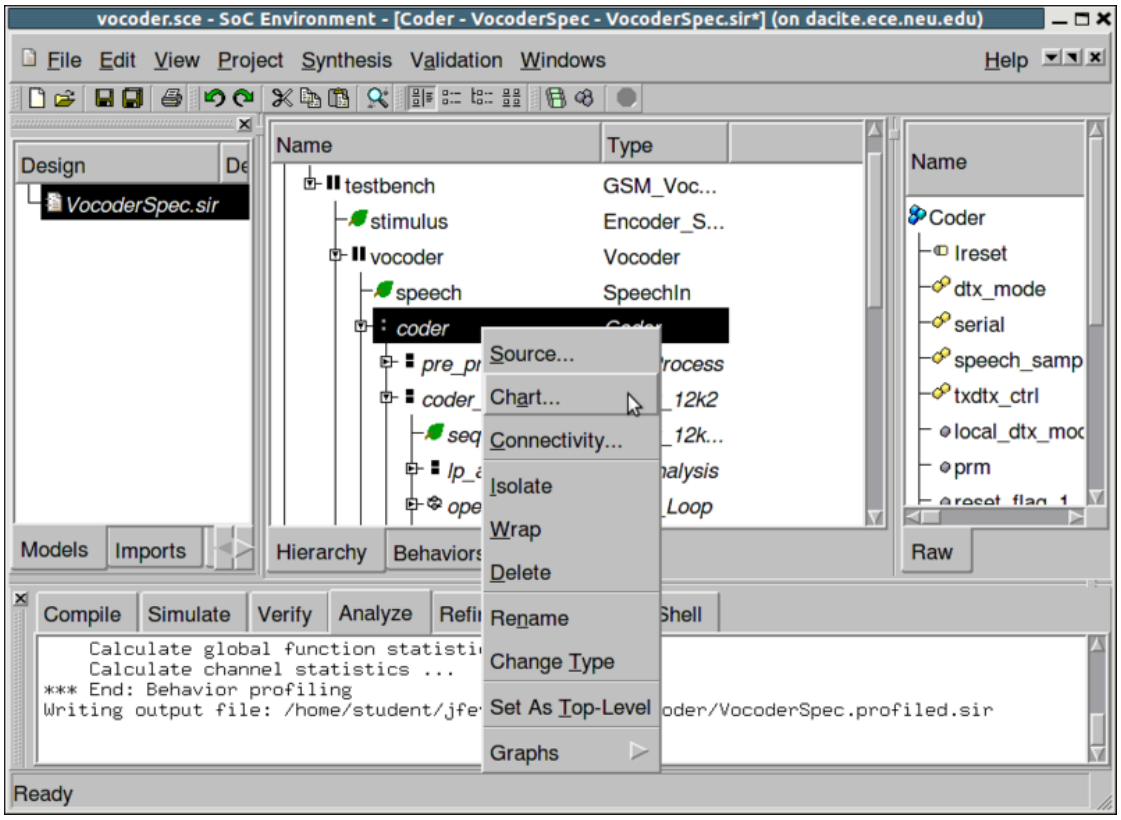
Note that a new window pops up in the design management area. It has two sub-windows. The sub-window on the left shows the Vocoder design hierarchy. The leaf behaviors are shown with a leaf icon next to them. For instance, we see two leaf behaviors: "stimulus", which is used to feed the test vectors to the design, and "monitor", which validates the response. "coder" is the top behavior of the Vocoder model. It can be seen from the icon besides the "coder" behavior that it is an FSM composition. This means the Vocoder specification is captured as a finite state machine. Also note in the logging window that the SoC design has been compiled into an intermediate format. Upon opening a source file into the design window, it is automatically compiled into its unique internal representation files (SIR) which in turn is used by the tools that work on the model.

2.2.3.4. Open specification model (cont'd)



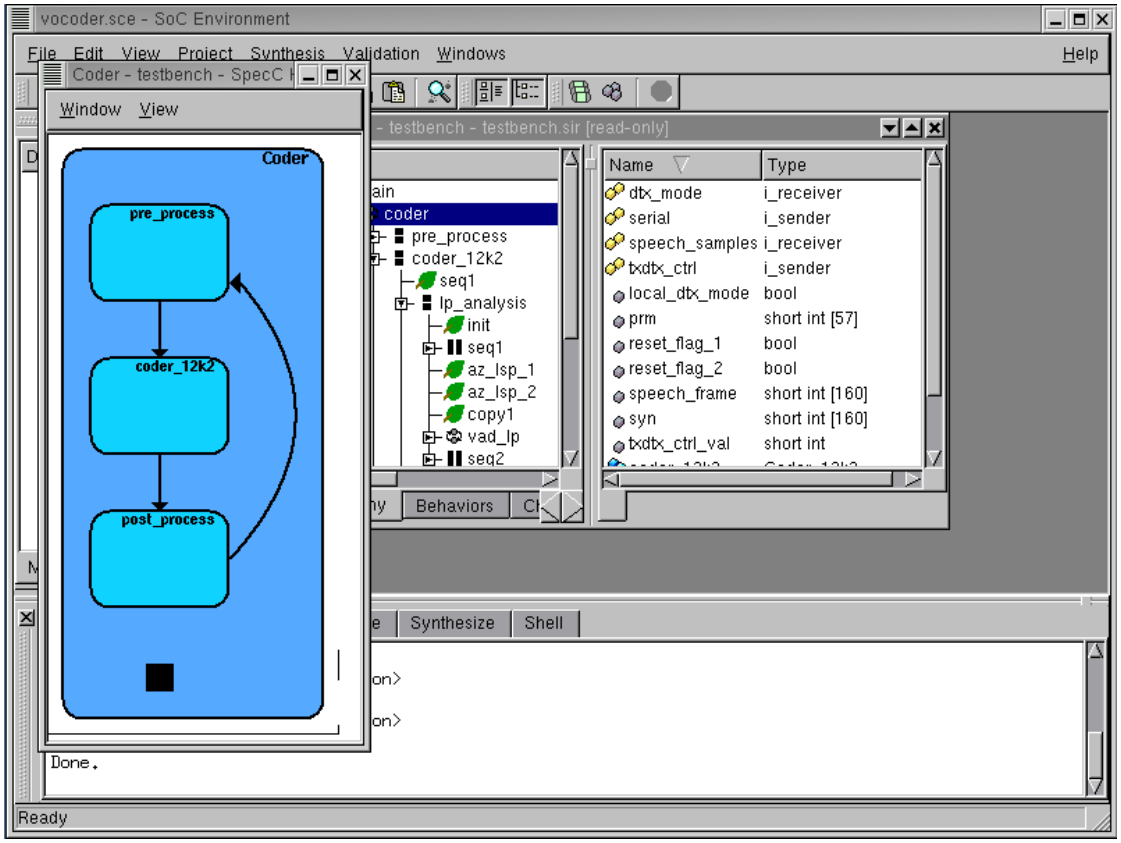
The model may be browsed using the design hierarchy window. Parallel composition is shown with || shaped icons and sequential composition with ':' shaped icons. On selecting a behavior in the design hierarchy window, we can see the behavior's characteristics in the right sub-window. For instance, the behavior "vad_lp" has ports shown with yellow icons, variables with gray icons and sub-behaviors with blue icons.

2.2.3.5. Open specification model (cont'd)



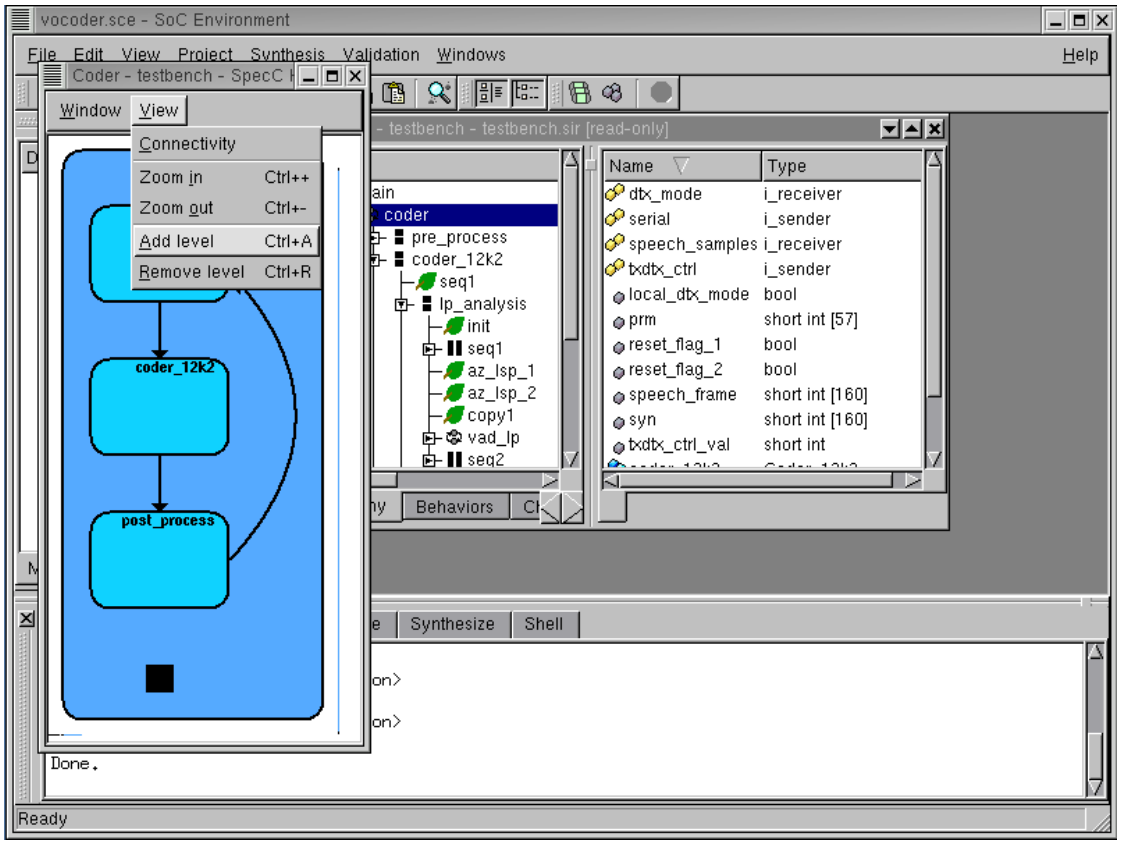
Before making any synthesis decisions, it is important to understand the composition of the specification model. It is useful because the composition really tells us which features of the model may be exploited to gain maximum productivity. Naturally, the most intuitive way to understand a model's structure is through a graphical representation. Since system models are typically very complex, it is more convenient to have a hierarchical view which may be easily traversed. SCE provides for such a mechanism. To graphically view the hierarchy, from the design hierarchy window, select "coder". Right click and select Hierarchy. Notice that the menu provides for a variety of services on individual behaviors. We shall be using one or more of these in due course.

2.2.4. Browse specification model



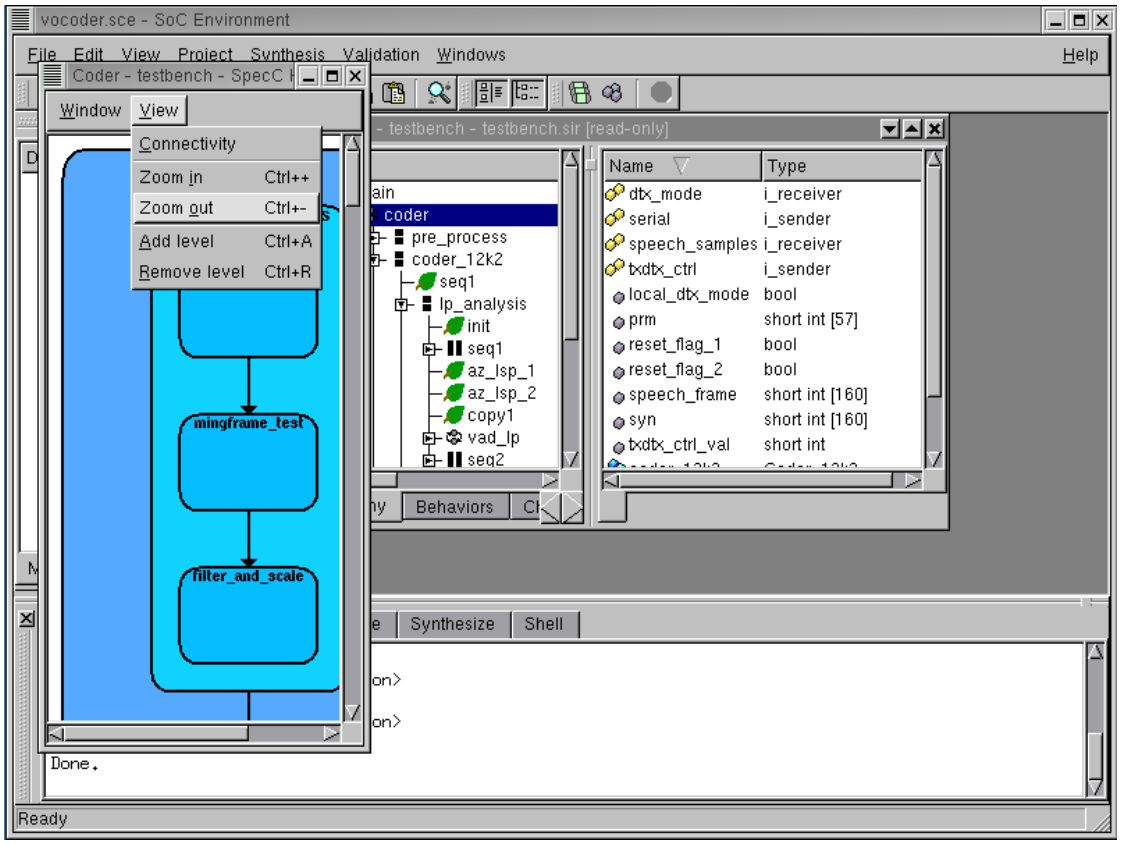
A new window pops up showing the Vocoder model in graphical form. As noted earlier, the specification is an FSM at the top level with three states of pre-processing, the bulk of the coder functionality itself and finally post-processing.

2.2.4.1. Browse specification model (cont'd)



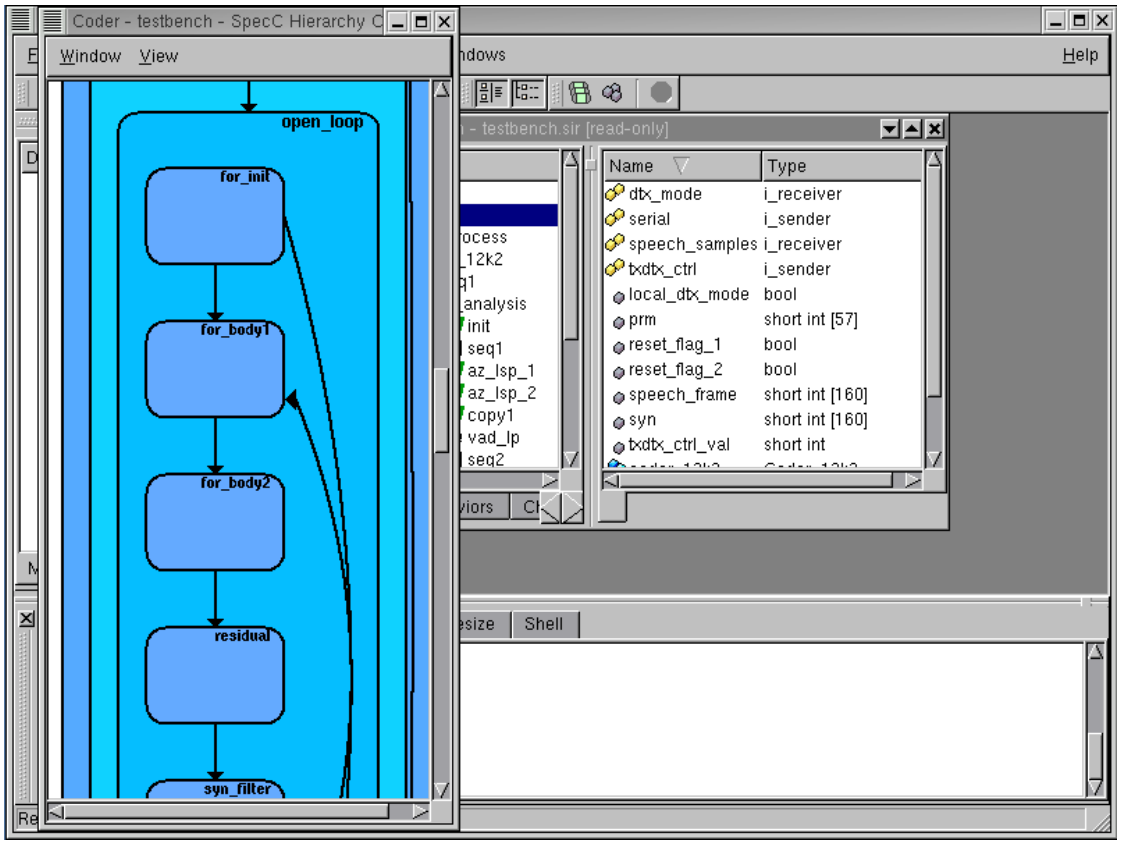
At this stage, we would like to delve into greater detail of the specification. To view the model graphically with higher detail, select View⟶Add level. Perform this action twice to get a more detailed view. As can be seen, the View menu provides features like displaying connectivity of behaviors, modifying detail level and zooming in and out to get a better view.

2.2.4.2. Browse specification model (cont'd)



Zoom out to get a better view by selecting View⟶Zoom out

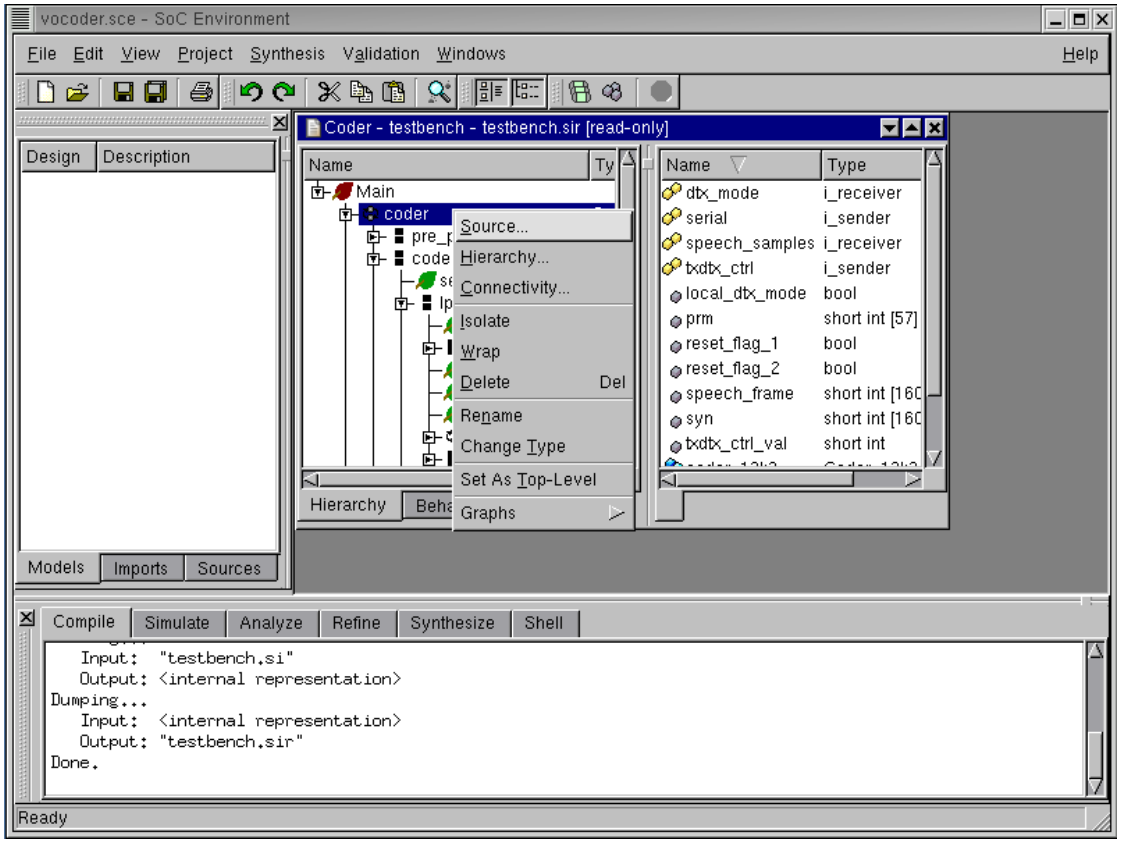
2.2.4.3. Browse specification model (cont'd)



Scroll down the window to see the FSM and sequential composition of the Vocoder model. Note that the specification model of the GSM Vocoder does not contain much parallelism. Instead, many behaviors are sequentially executed. This is due to the several data dependencies in the code. For our implementation, this is an important observation. Since there is not much parallelism in the code to exploit, speedup can be achieved only by use of faster components. One way to speed up is to use dedicated hardware units.

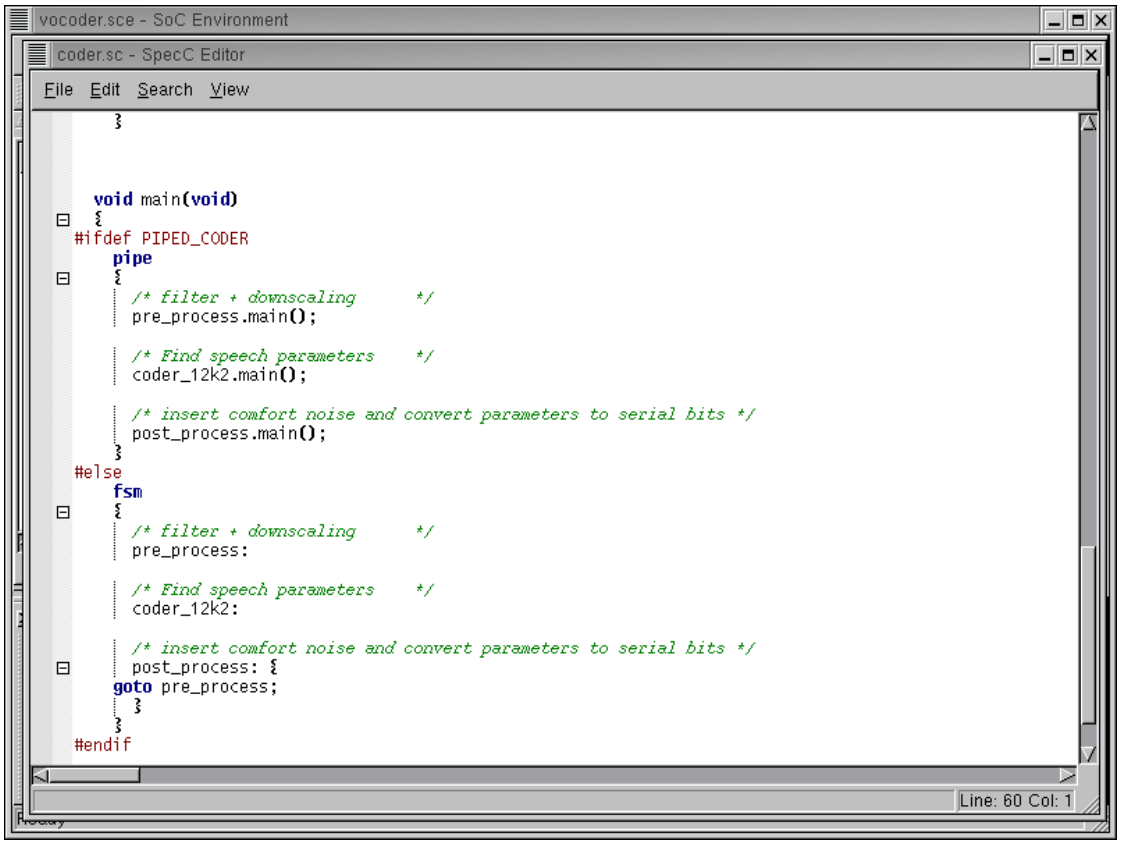
Exit the hierarchy browser by selecting Window⟶Close

2.2.5. View specification model source code



We can also view the source of the models conveniently in SCE. For example, to check the source for behavior "coder", just click on the row in the hierarchy to select it. Then right click to bring up a menu and click on Source.

2.2.5.1. View specification model source code(cont'd)



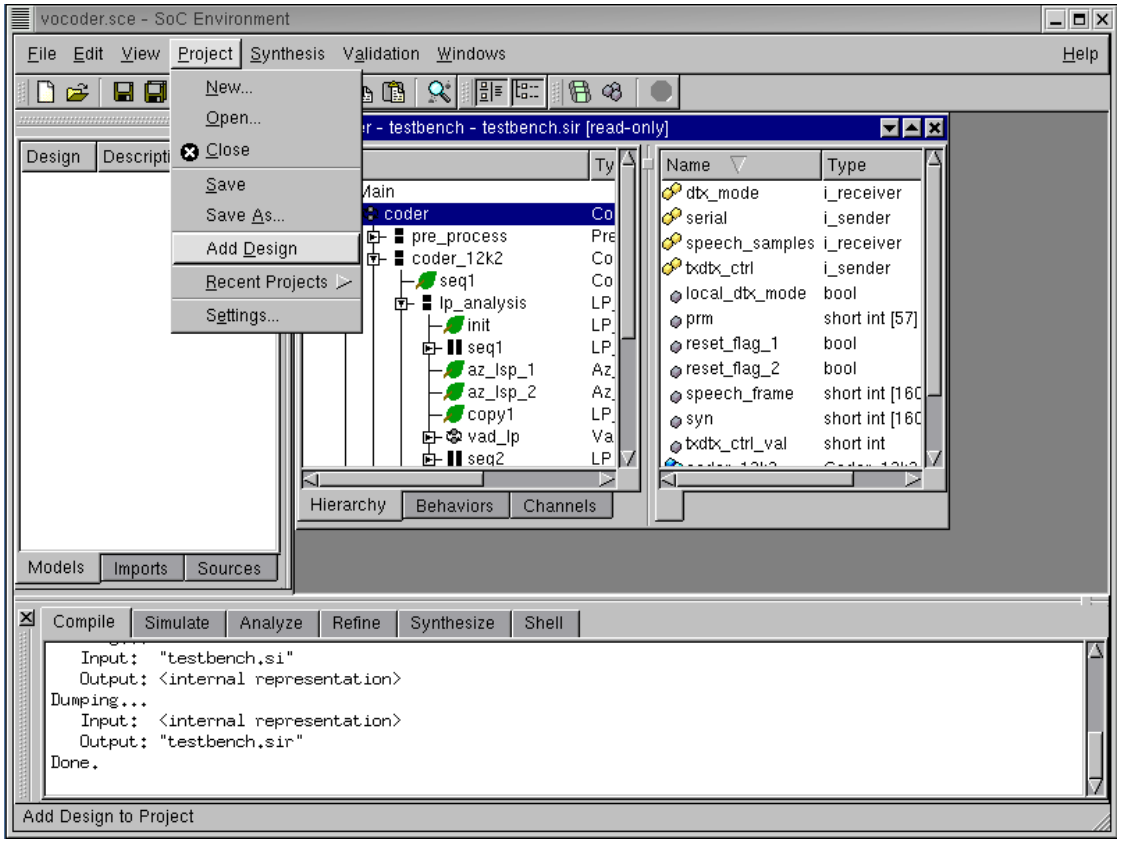
The SpecC Editor pops up containing the source code for the selected behavior. Changes to the source code can be made using the editor. After reviewing the source code, close the editor by selecting File⟶Close from its menu bar.

2.3. Simulation and Analysis

Once we have captured the specification as a model and browsed through its behavioral hierarchy and connectivity, we need to ensure that our specification is correct. We also need to analyze our specification model to derive interesting observations about the nature of the computation. The check for correctness is done by simulating the model. Note that the model is purely functional, so the simulation runs very quickly. This is also a good time to debug the model for functional errors that might have crept in while writing it.

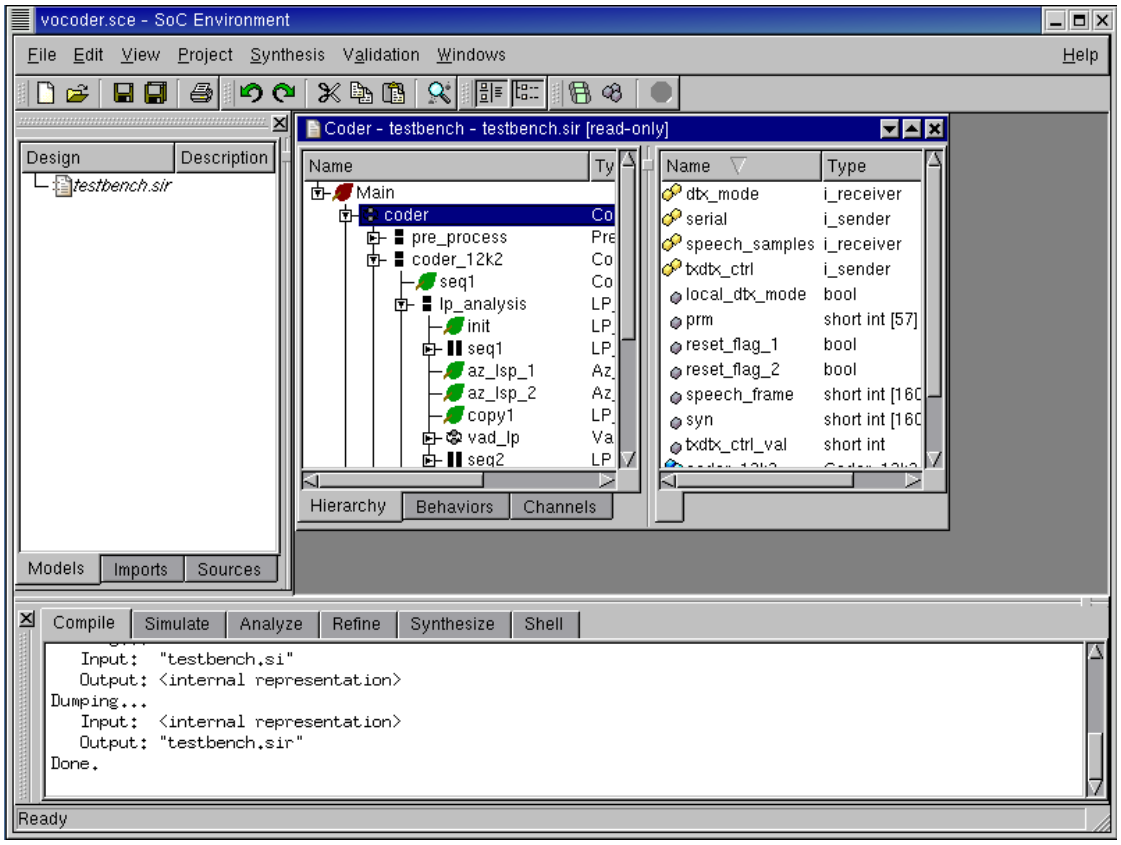
After the model is verified to be functionally correct, we proceed to the analysis phase. For this, we need to profile the model using the profiling tool available in SCE. The profile gives us useful information like the about of computation, its distribution over the various behaviors in the model and its nature. This information is need to make crucial architectural choices as we will see as the demo proceeds.

2.3.1. Simulate specification model



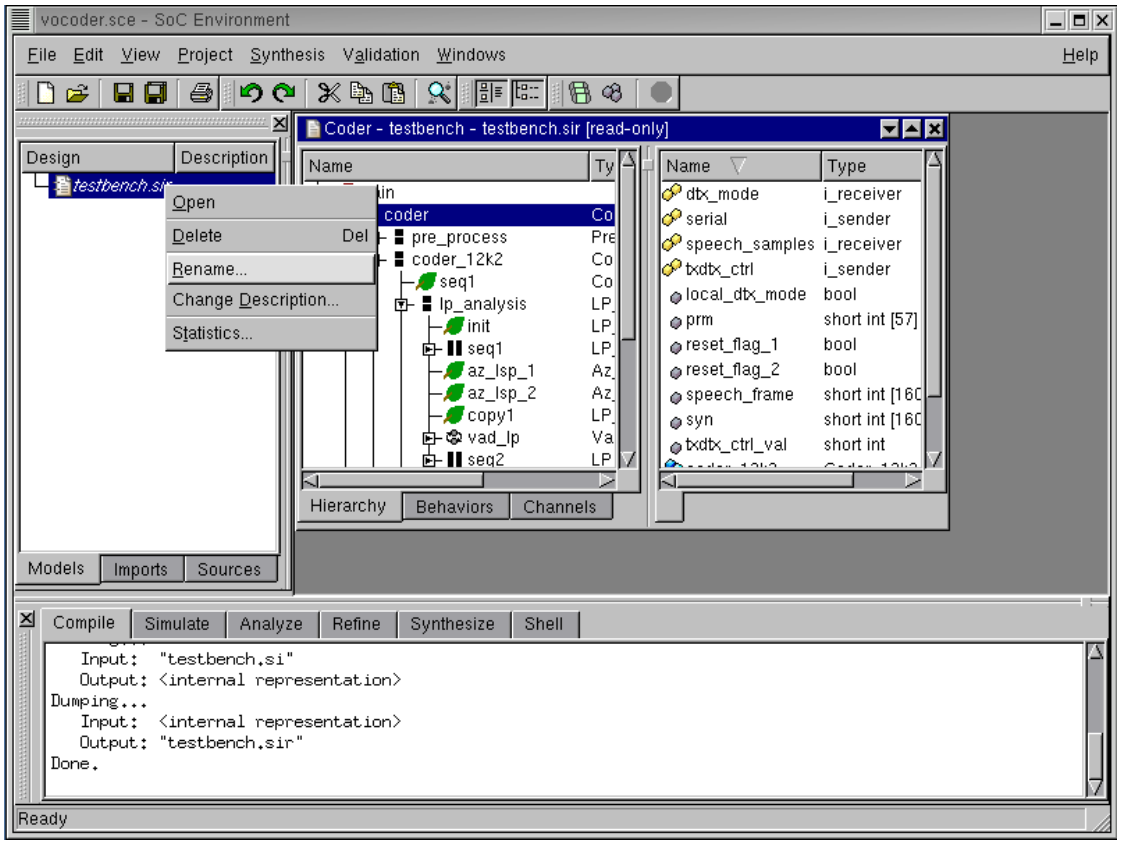
We must now proceed to validate the specification model. Remember that we have a "golden" output for encoding of the 163 frames of speech. The specification model would meet its requirements if we can simulate it to produce an exact match with the golden output. In practice, a more rigorous validation process is involved. However, for the purpose of the tutorial, we will limit ourselves to one simulation only. Start with adding the current design to our Vocoder project by selecting Project⟶Add Design from the menu bar.

2.3.1.1. Simulate specification model (cont'd)



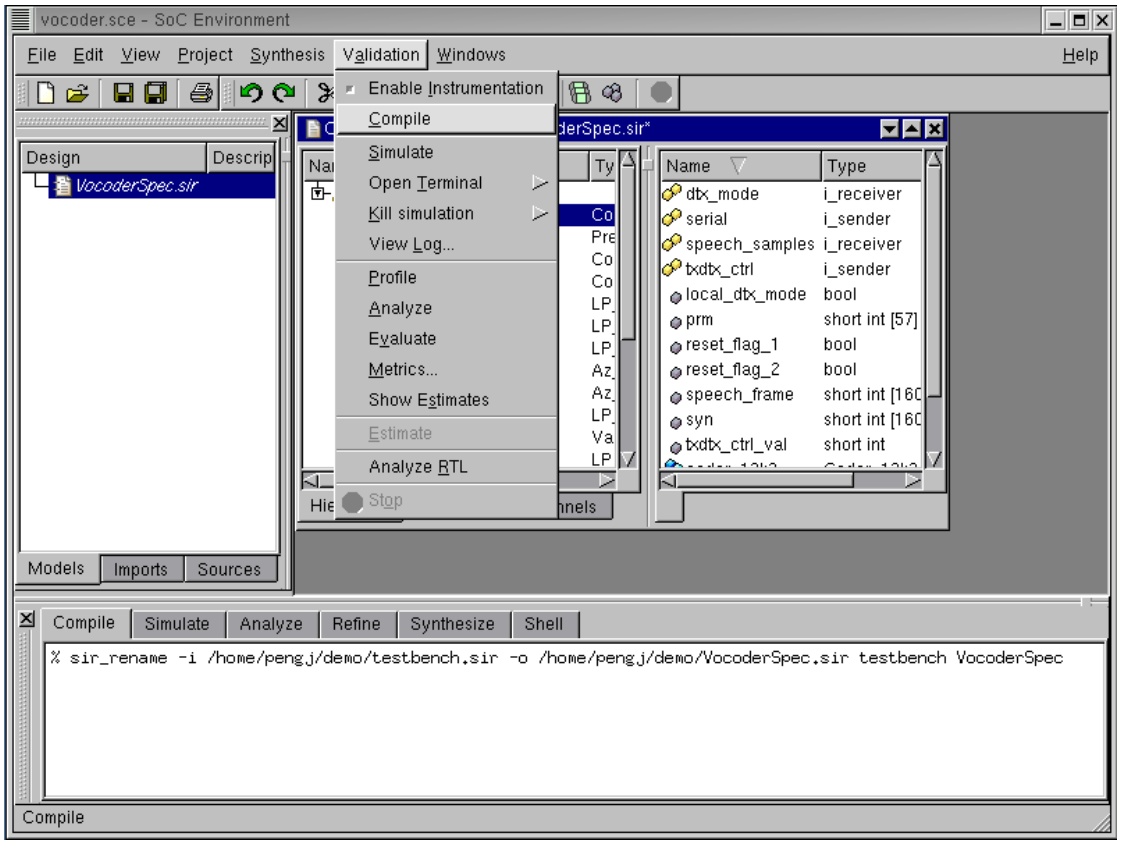
The project is now added as seen in the project management workspace on the left in the GUI.

2.3.1.2. Simulate specification model (cont'd)



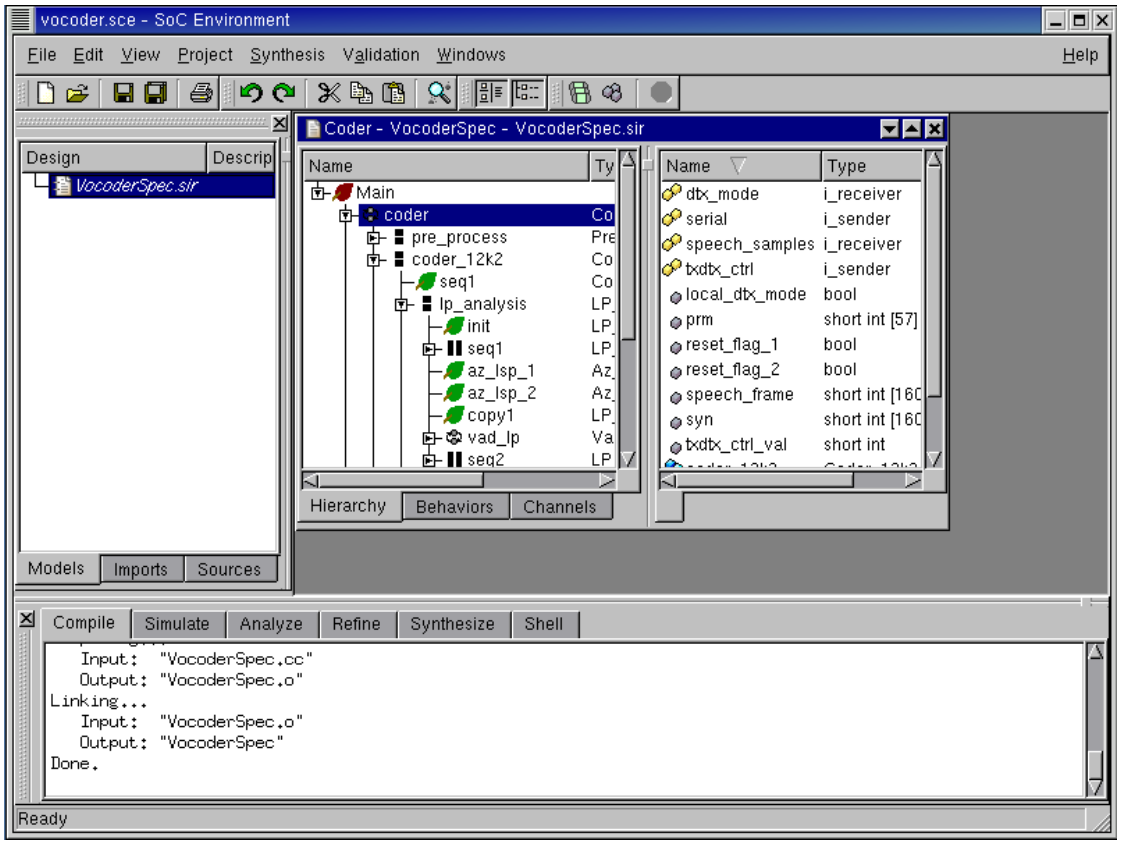
We must now rename the project to have a suitable name. Remember that our methodology involved 4 models at different levels of abstraction. As these new models are produced, we need to keep track of them. Right click on "testbench.sir" and select Rename to rename the design to "VocoderSpec". This indicates that the current model corresponds to the topmost level of abstraction, namely the specification level. Note that the extension ".sir" would be automatically appended. Also note that a model may be made activated, deleted, renamed and and its description modified by right click on its name in the project management window.

2.3.1.3. Simulate specification model (cont'd)



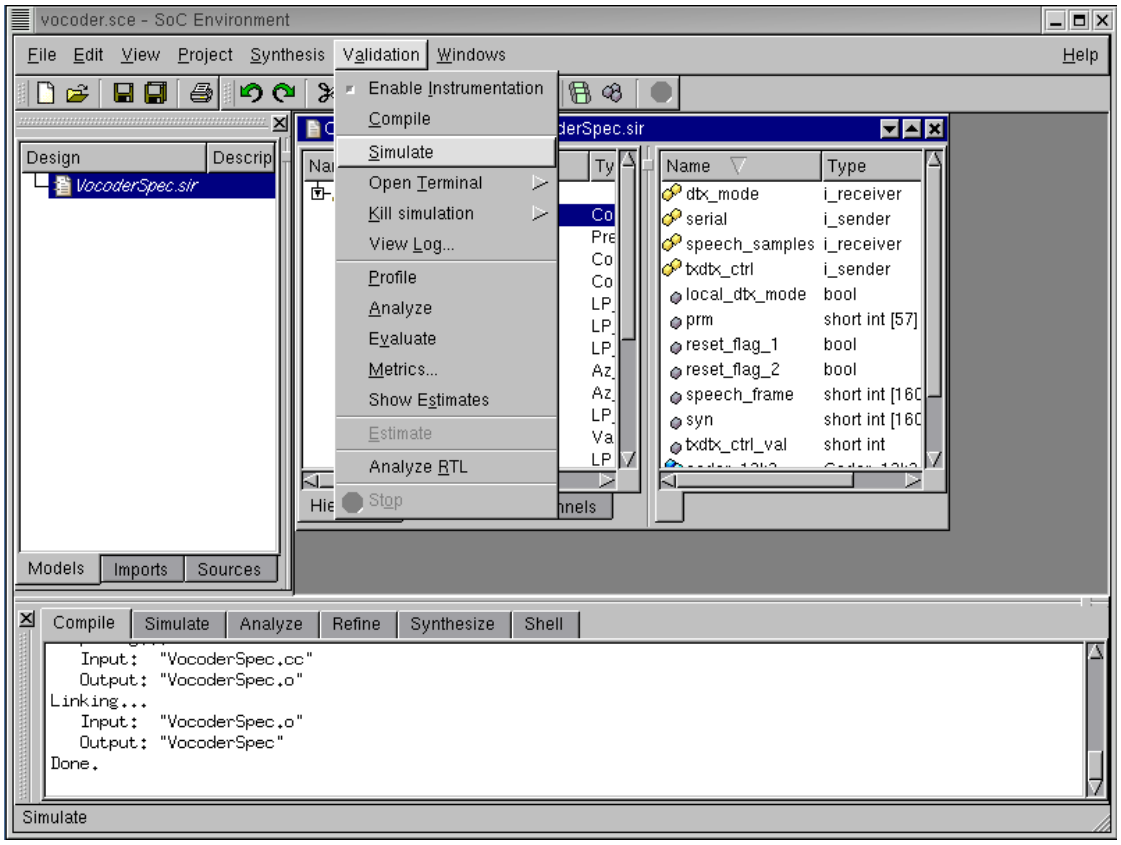
After the project is renamed to "VocoderSpec.sir", we need to compile it to produce an executable. This may be done by selecting Validation⟶Compile from the menu bar. Note that the validation menu also provides for code instrumentation which is used for profiling. Moreover, we have choices for simulating the model, opening a simulation terminal, killing a running simulation, viewing the log, profiling, analyzing simulation results, model evaluation, displaying metrics and estimates etc. All these features will be used in due course of our system design process.

2.3.1.4. Simulate specification model (cont'd)



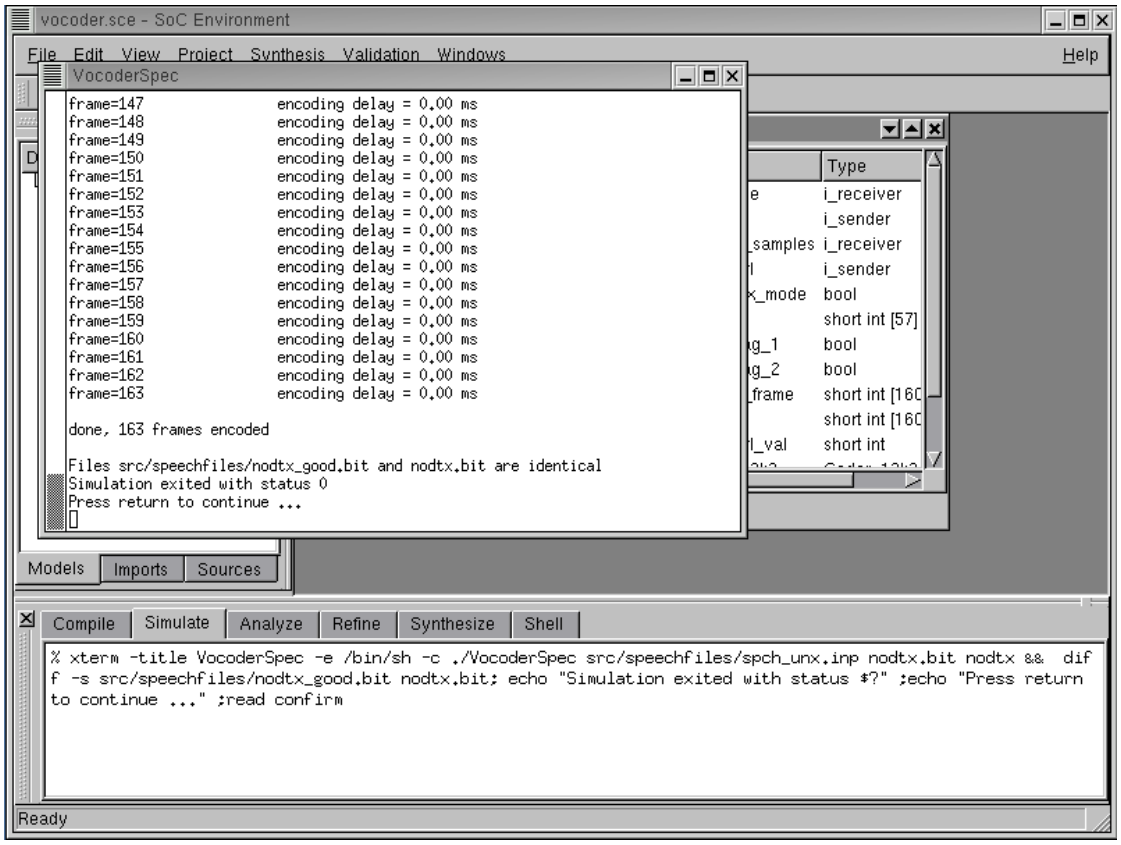
Note that in the logging window we see the compilation messages and an output executable "VocoderSpec" is created.

2.3.1.5. Simulate specification model (cont'd)



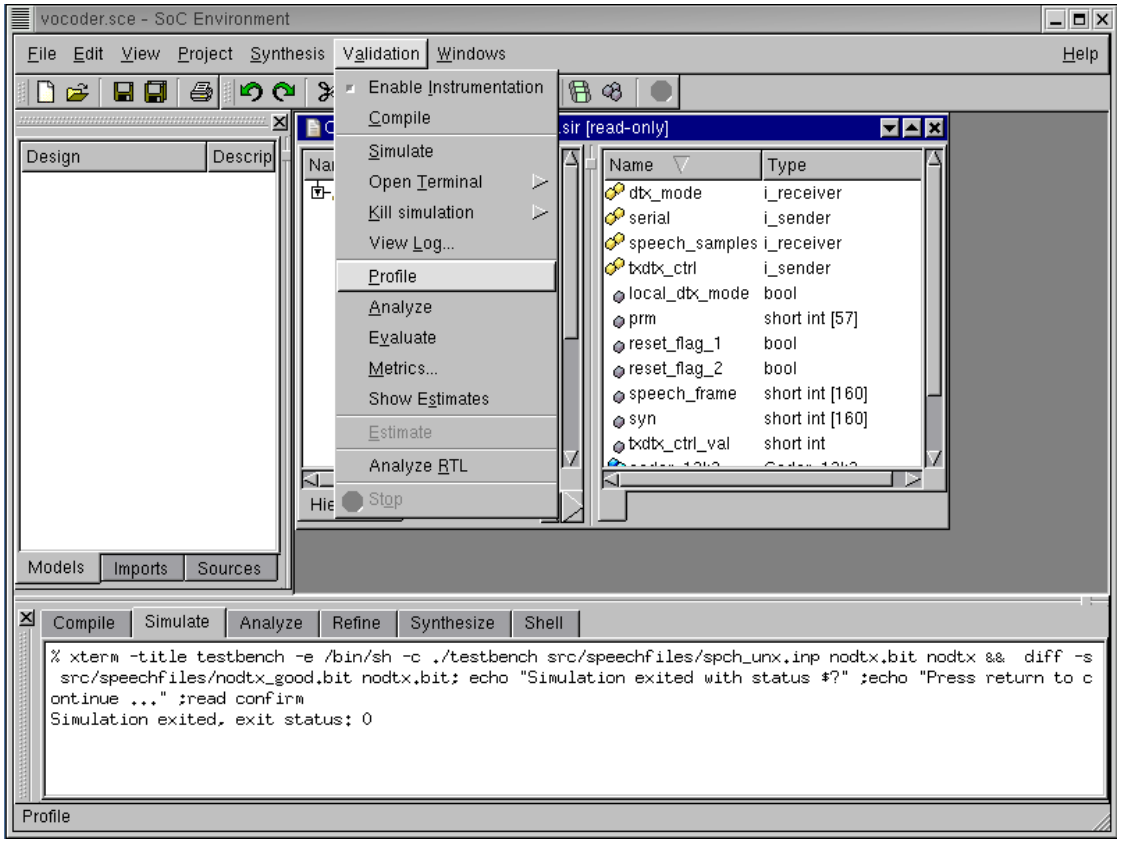
The next step is to simulate the model to verify whether it meets our requirements or not. This may be done by selecting Validation⟶Simulate from the menu bar.

2.3.1.6. Simulate specification model (cont'd)



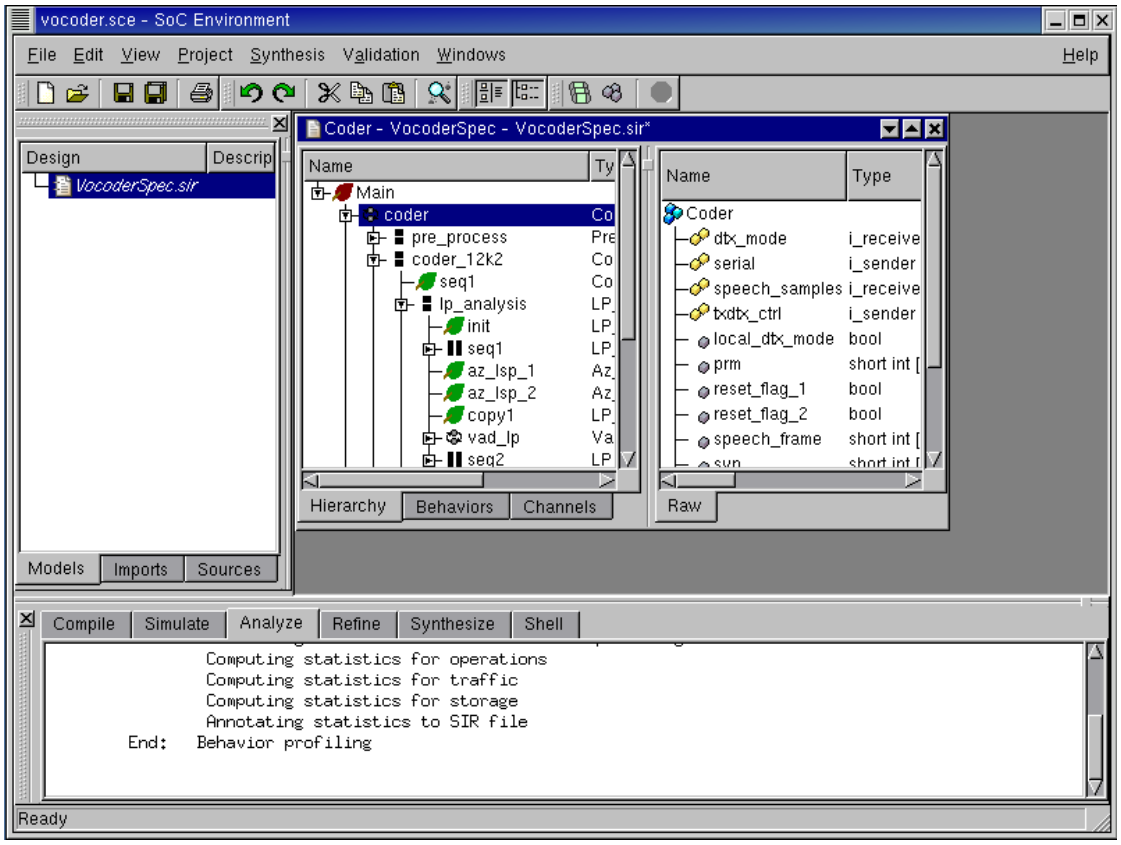
Note that an xterm pops up showing the simulation of the Vocoder specification model on a 163 frame speech sample. The simulation should finish correctly which is indicated by the exit status being '0'. It can be seen that 163 speech frames were correctly simulated and the resulting bit file matches the one given with the vocoder standard. It may be noted that each frame has an encoding delay of 0 ms. This is a because our specification model has no notion of timing. As explained in the methodology, the specification is a purely functional representation of the design and is devoid of timing. For this reason, all behaviors in the model execute in 0 time thereby giving an encoding delay of 0 for each frame. Press RETURN to close this window and proceed to the next step.

2.3.2. Profile specification model



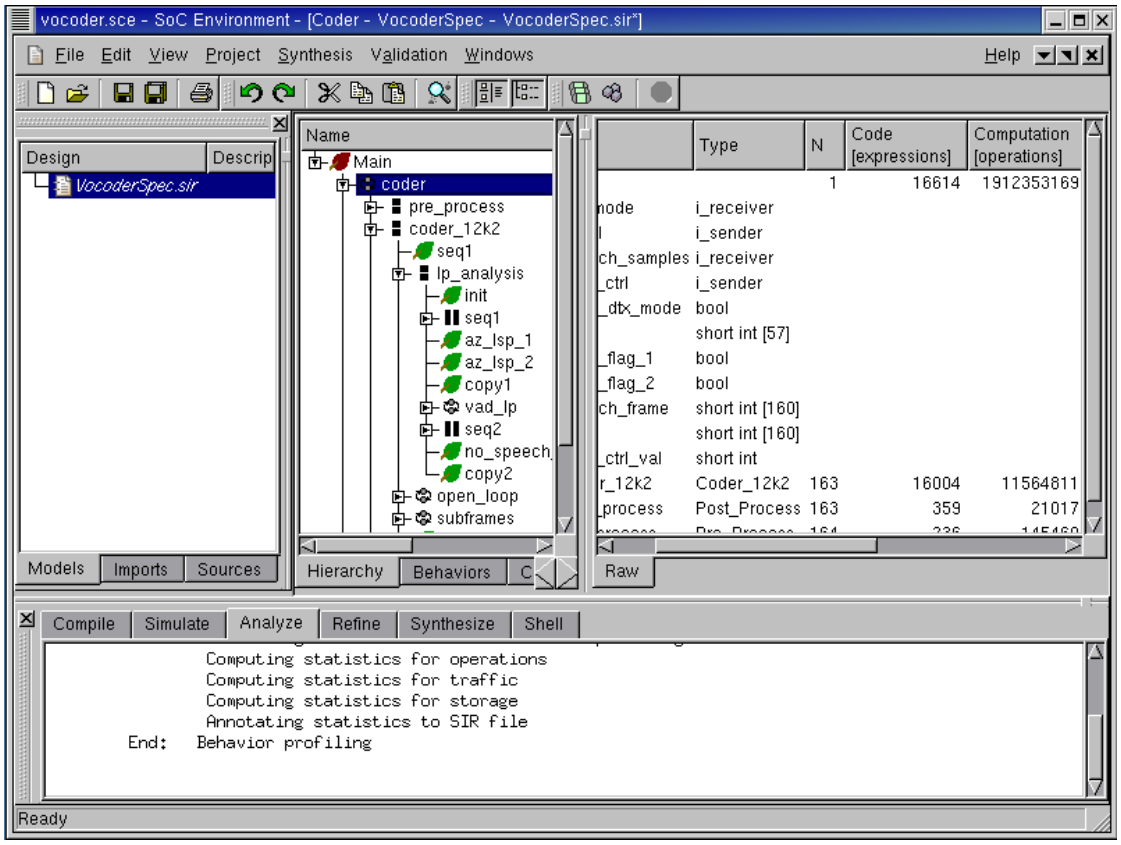
In order to select the right architecture for implementing the model, we must begin by profiling the specification model. Profiling provides us with useful data needed for comparative analysis of various modules in the design. It also counts the various metrics like number of operations, class and type of operation, data exchanged between behaviors etc. These statistics are collected during simulation. Profiling may be done by selecting Validation⟶Profile from the menu bar.

2.3.2.1. Profile specification model (cont'd)



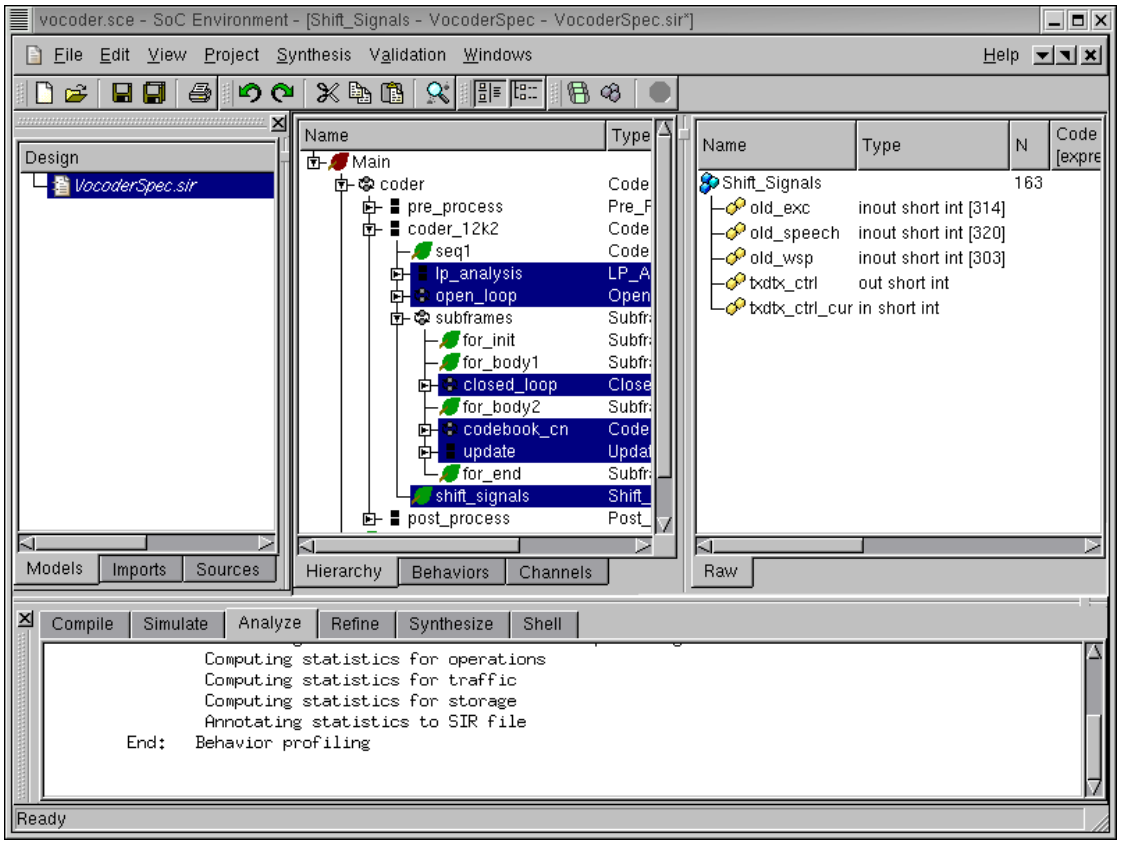
The logging window now shows the results of the profiling command. Note that there is a series of steps for computing statistics for individual metrics like operations, traffic, storage etc. Once these statistics are computed, they are annotated to the model and displayed in the design window.

2.3.2.2. Profile specification model (cont'd)



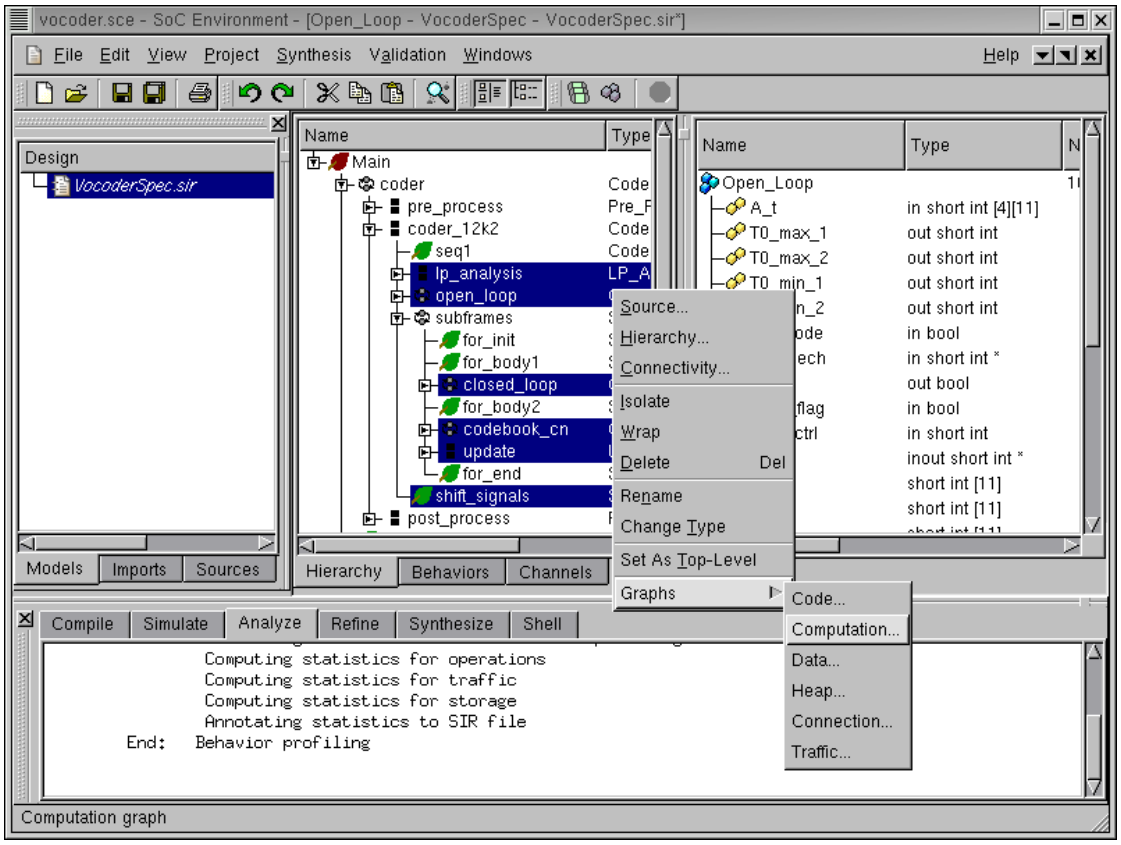
It may also be noted that the design management window now has new column entries that contain the profile data. Maximize this window and scroll to the right to see various metrics for behaviors selected in the design hierarchy. The current screen shot shows Computation, Data, Connections and Traffic for the top level behavior "coder". Computation essentially means the number of operations in each of the behaviors. Data refers to the amount of memory required by the behaviors. Connections indicate the presence of inter-behavior channels or connection through variables. Traffic refers to the actual amount of data exchanged between behaviors. The metrics may also be obtained for other behaviors in the design besides "coder".

2.3.3. Analyze profiling results



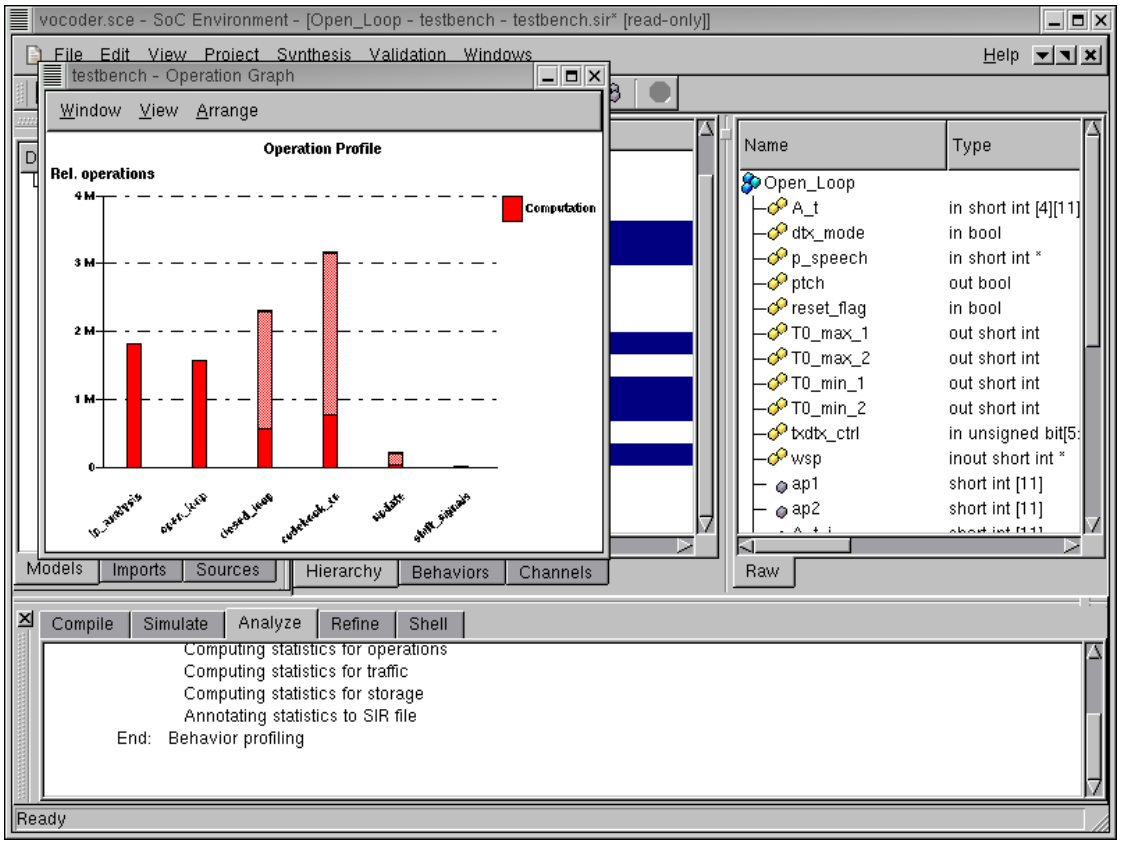
Once we have the profiling results, we need a comparative analysis of the various behaviors to enable suitable partitioning. Here we analyze the six most computationally intensive behaviors namely "lp_analysis", "open_loop", "closed_loop", "codebook_cn", "update" and "shift_signals." They may be multi-selected in the design hierarchy by pressing CNTRL key and left clicking on them. These particular behaviors were selected because these are the major blocks in the behavior "coder_12k2", which in turn is the central block of the entire coder. Thus the selected behaviors show essentially the major part of the activity in the coder. We ignore the pre-processing and the post-processing blocks, because they are of relatively lower importance.

2.3.3.1. Analyze profiling results (cont'd)



In order to select a suitable architecture for implementing the system, we must perform not only an absolute but also a comparative study of the computation requirements of the selected behaviors. SCE provides for graphical view of profiling statistics which may be used for this purpose. After the multi-selection, we right click and select Graphs⟶Computation from the menu bar.

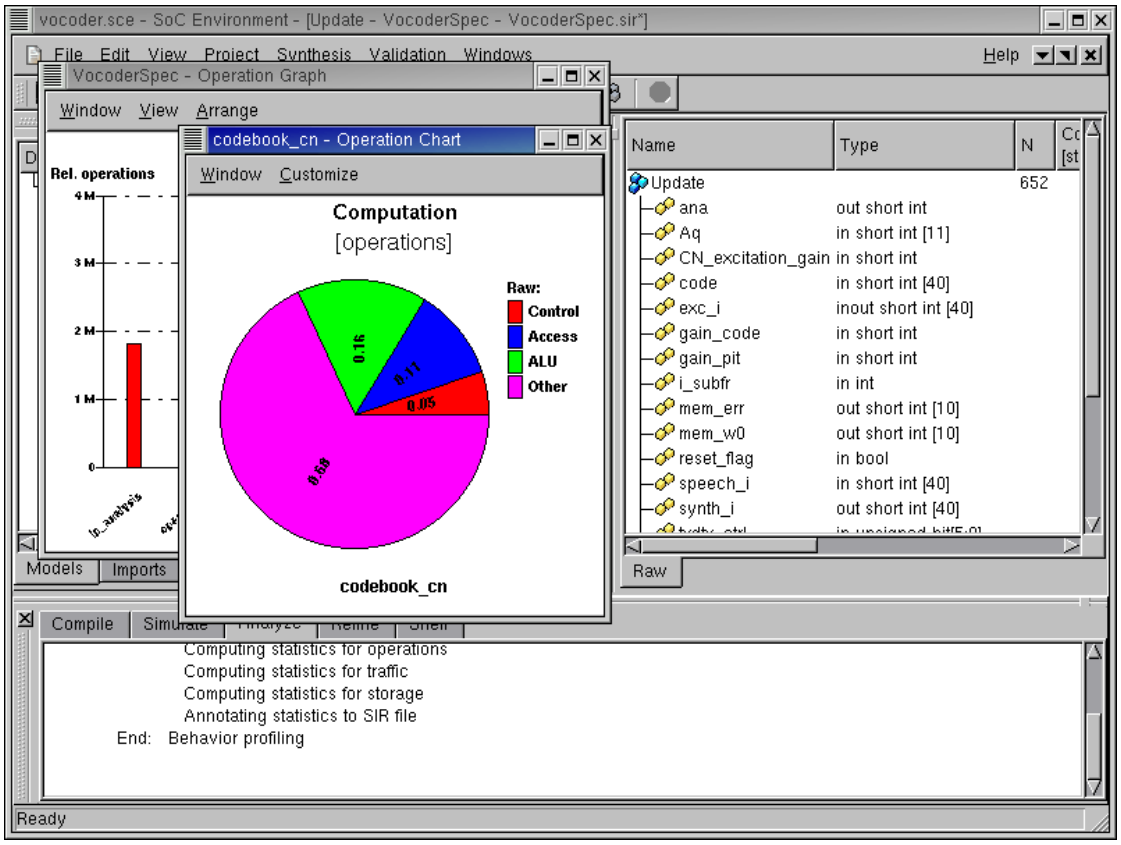
2.3.3.2. Analyze profiling results (cont'd)



We now see a bar graph showing the relative computational intensity of the various behaviors in the selected behaviors. Essentially, the graph shows the number of operations on the Y-axis for the individual behaviors on the X-axis. Double click on the bar for codebook_cn to view the distribution of its various operations. Note that we select "codebook_cn" because it is the behavior with the most computational complexity.

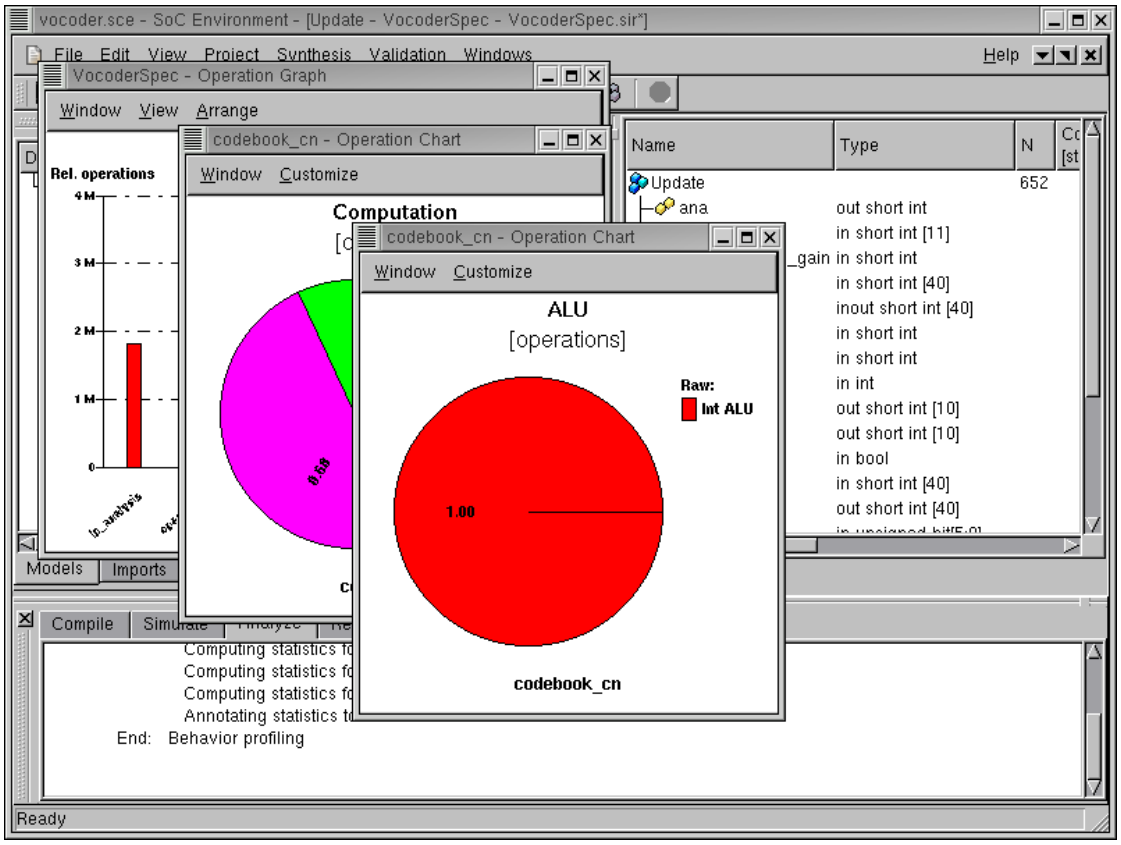
Note that the bars representing the computation for "codebook_cn" and "closed_loop" have two sections. The lower section is filled with red color and the upper section is partially shaded. Each speech frame consists of four sub-frames and the behaviors "codebook_cn" and "closed_loop" are executed for each subframe in contrast to other behaviors in the graph, which are executed once. Hence the filled section of the bar represents computation for each execution of behavior and the complete bar (including the shaded section) represents computation for the entire frame.

2.3.3.3. Analyze profiling results (cont'd)



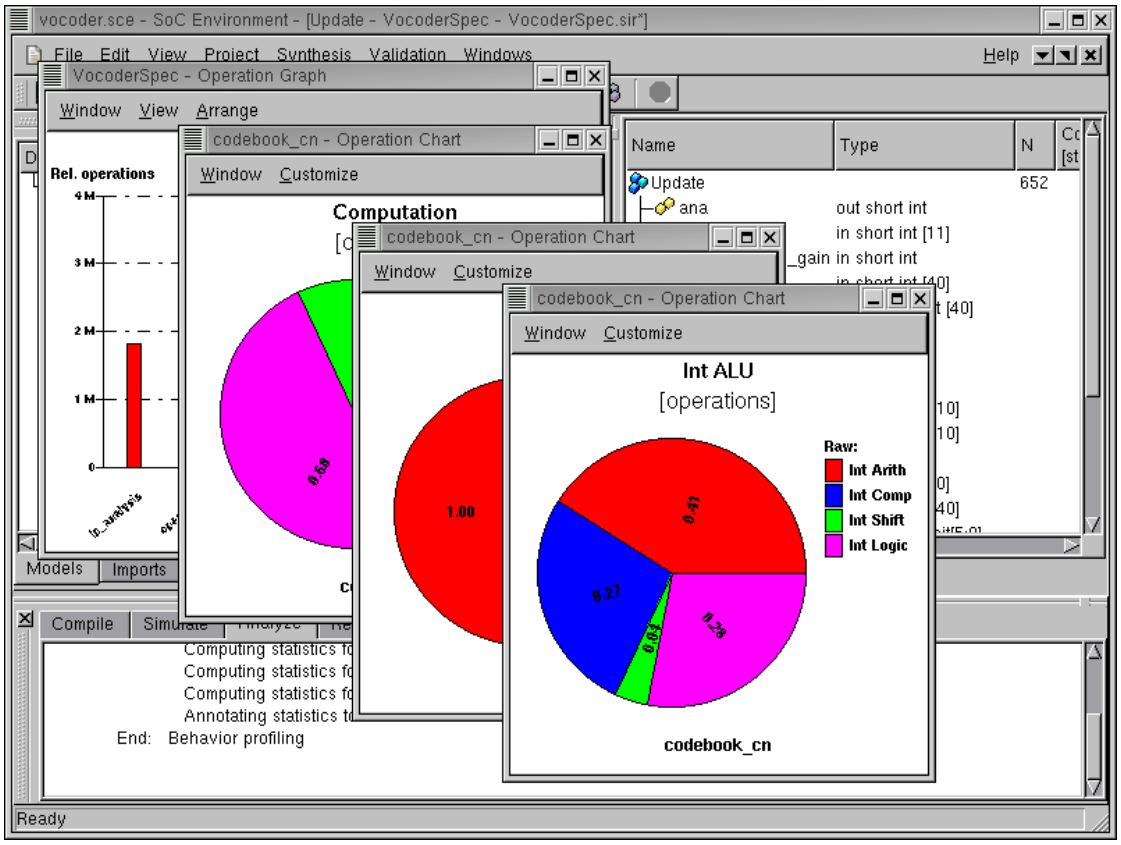
A new window pops up showing a pie chart. This pie chart shows the distribution of various operations like ALU, Control, Memory Access etc. We are interested in seeing the types of ALU operation for this design. To do this double click on the ALU (green) sector of the pie chart.

2.3.3.4. Analyze profiling results (cont'd)



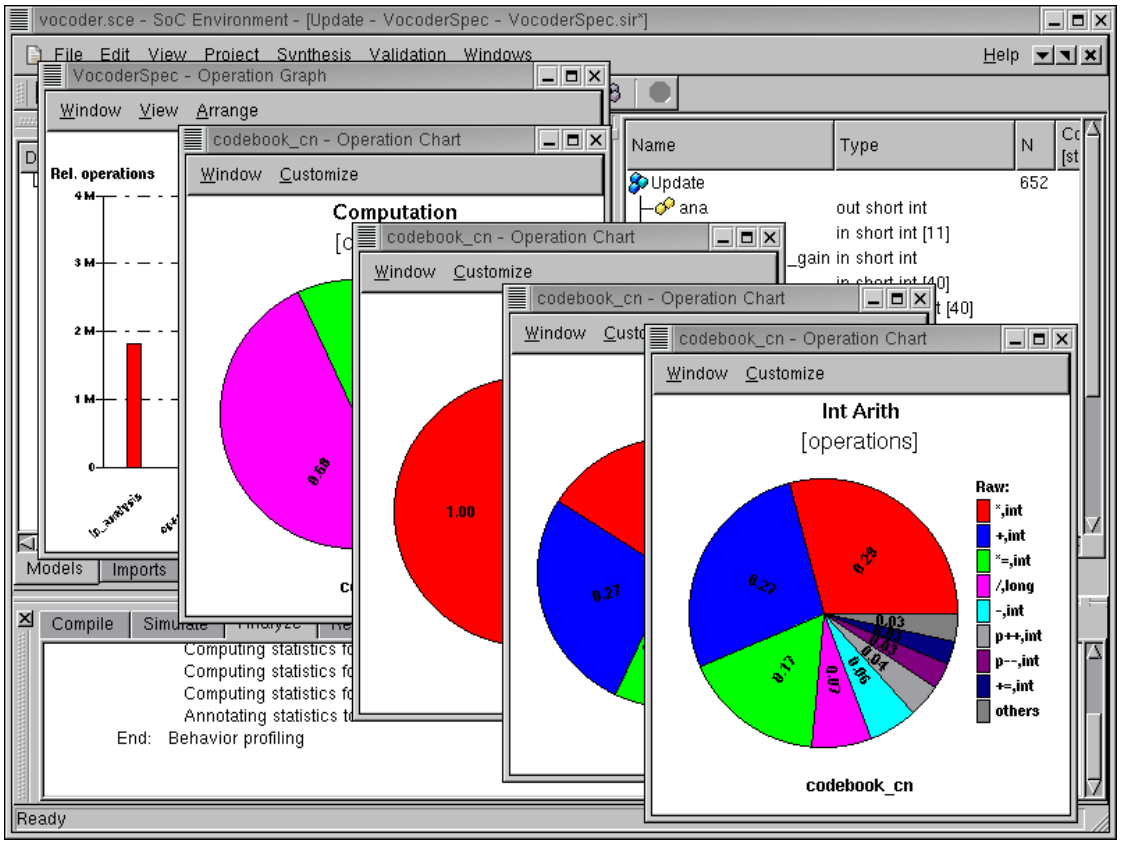
A new window pops up showing another pie chart. This pie chart shows the distribution of ALU operations. It can be seen that all the operations are integer operations, which is typical for signal processing application like the Vocoder. Since all the operations are integral, it does not make sense to have any floating point units in the design. Instead, we need a component with fast integer arithmetic like a DSP. To see the distribution of these integer operations, again double click on the pie chart.

2.3.3.5. Analyze profiling results (cont'd)



A new window pops up showing another pie chart. This pie chart shows the distribution of the type of integer operations. We can see that the majority of the operations is integer arithmetic. To view the distribution of the arithmetic operation types, again double click on the sector for "Int Arith".

2.3.3.6. Analyze profiling results (cont'd)



We can now observe the distribution of arithmetic operations like "multiplication", "addition", "increment", "decrement", etc. on a new pie chart. Note that 3 quarters of the operations are additions or multiplications, thus it would be a good idea to have these two operations directly supported by a specific hardware unit.

The combination of visual aids like bar graphs and pie charts gives a good idea of the nature of intended system. Please close all the pop-up windows to conclude the specification analysis phase.

2.4. Summary

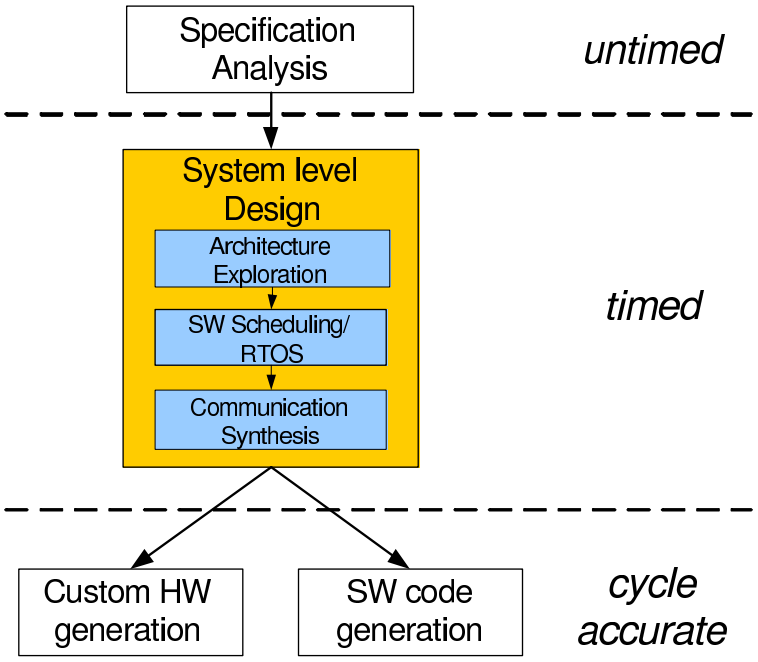
In this chapter we looked at how to start with the system specification and analyze its characteristics. We were familiarized with the SCE graphical user interface and the profiling, analysis and simulation tools. By means of graphical tools, we were able to traverse the hierarchy of the system specification model. Graphical representations also provided us with information on connectivity between behaviors in the design. The user friendliness of these representations allows us to analyze our design better which would otherwise be very cumbersome.

Profiling and statistical data about the specification model also gives us interesting hints. For instance, the nature of computation in the model shows us the appropriate components to consider for the system architecture. Similarly, pie charts and bar graphs for the distribution of computation show us the critical behaviors and their nature. As we move forward in the system design process, we will have to make design decisions at various stages and such statistical analysis will be of great value. In future implementations on the tool, these analysis results may even be fed to automatic tools to generate optimal system architectures.

Chapter 3. System Level Design

3.1. Overview

Figure 3-1. System level design phase using SCE



System design is increasingly being performed at higher levels of abstraction to deal with a variety of issues. In this chapter, we look at system level design tasks with SCE as highlighted in Figure 3-1. Firstly, we need to deal with both HW and SW in a single model. Secondly, and more importantly, complexity becomes unmanageable. In this chapter we will look at the system level design phase as shown in the above figure. This phase comprises of architecture exploration, serialization/RTOS insertion and communication synthesis. Architecture exploration deals with coming up with a suitable system architecture and distributing the system tasks in the specification onto those components. Since each component has a single control, we need to serialize the tasks in each component. Tasks that are mapped to SW can be dynamically scheduled on the processor by inserting an RTOS model. Finally, we perform communication synthesis to come up with a communication architecture and refine the data transfer and interfaces to use the

communication architecture. The goal of this phase is to come up with a model that can serve as an input to RTL synthesis for HW components and SW generation for processors.

3.2. Architecture Exploration

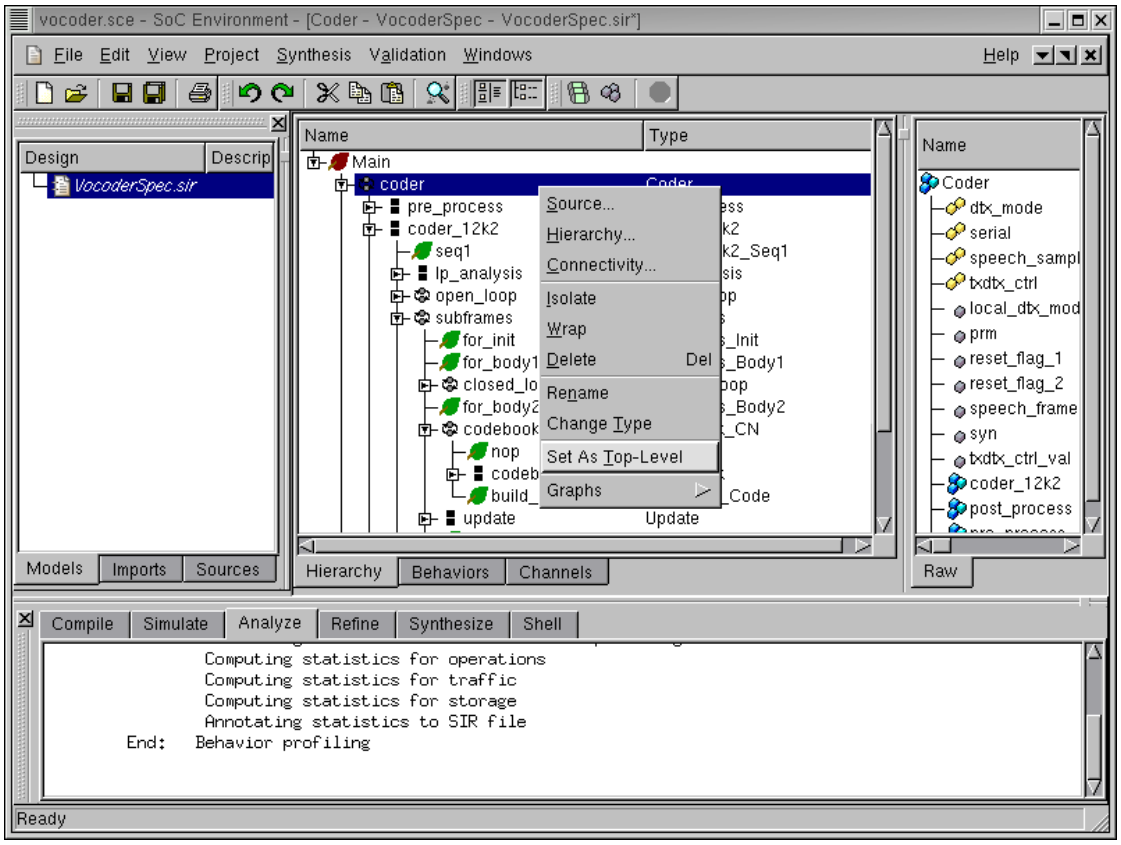
Architecture exploration is the design step to find the system level architecture and map different parts of the specification to the allocated system components under design constraints. It consists of the tasks of selecting the target set of components, mapping behaviors to the selected components and implementing correct synchronization between the components. Note that the components themselves are independent entities that execute in a parallel composition. In order to maintain the original semantics of the specification, the components need to be synchronized as necessary. Architecture exploration is usually an iterative process, where different candidate architectures and mappings are experimented to search for a satisfactory solution.

As indicated earlier, the timing constraint for the Vocoder design is the real time response requirement, i.e., the time to encode and decode the speech should be less than the speech time. The test speech has a 3.26 seconds duration. Therefore, the final implementation must meet this time constraint. In this chapter we see how we arrive at a suitable architecture with keeping this requirement in mind and using the refinement tool.

3.2.1. Try pure software implementation

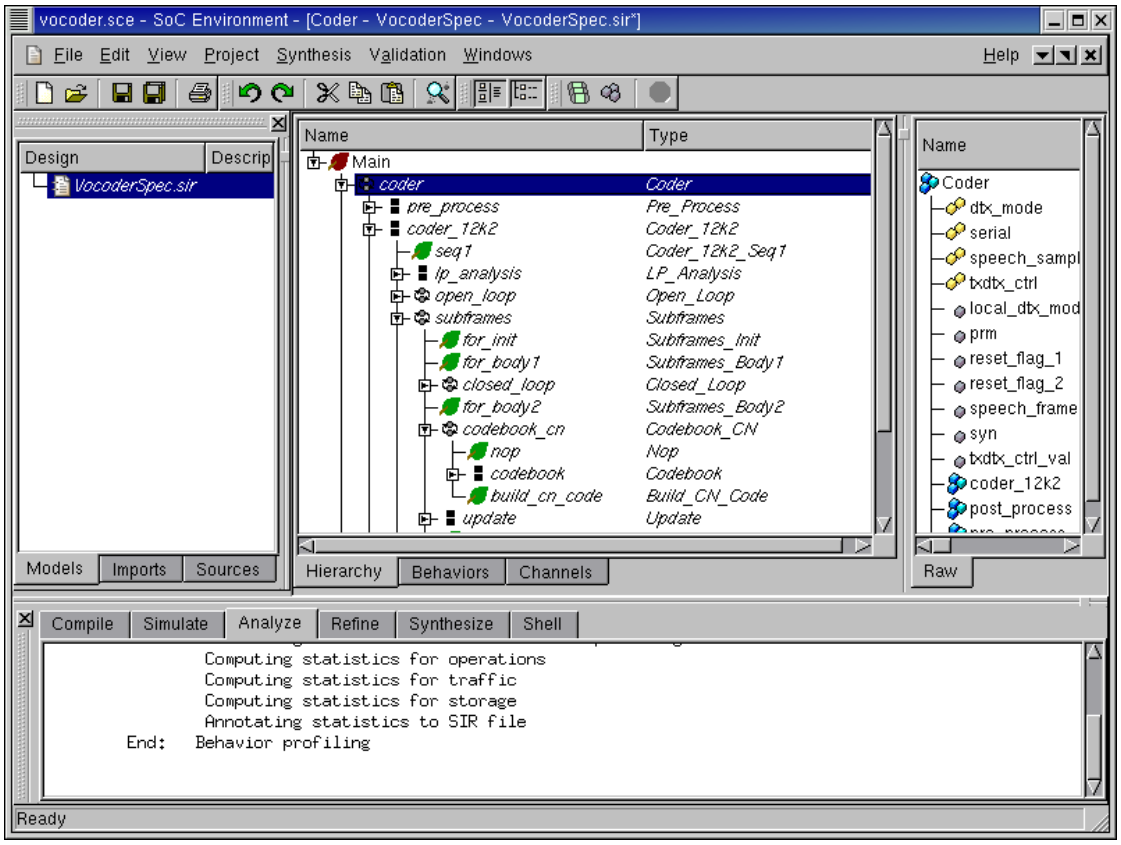
The goal of our exploration process is to implement the given functionality on a minimal cost architecture and still meet the timing constraint. The first approach is to implement everything in software so that we do not have the overhead of adding extra hardware and associated interfaces. To accomplish this, we first select a processor out of our component database. Thereafter, we map the entire specification on to this processor. Once the mapping is done, we invoke the analysis tool to see if the processor alone is sufficient to implement the system.

3.2.1.1. Try pure software implementation (cont'd)



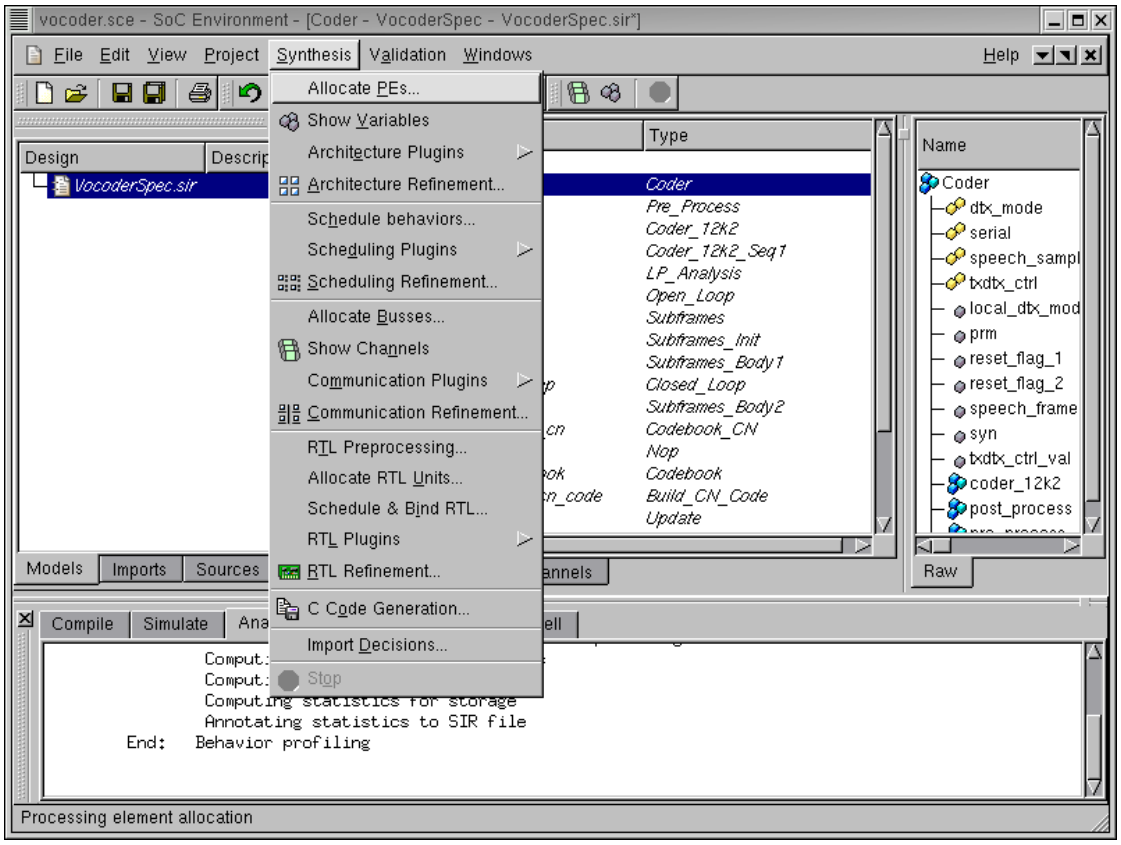
Before we move on, the top level behavior of the design needs to be specified. This is necessary because the specification model may have some test bench behaviors, which are not going to be included in the final design. It may be recalled that the project we are working with involves not only the design-under-test (DUT) but also the behaviors that drive it. For example, the behaviors "Monitor" and "Stimulus" are just testbench behaviors while the behavior "Coder" represents the real design. To specify "Coder" as the top level behavior, right click on "Coder" to bring up a drop box menu then left click on Set As Top-Level.

3.2.1.2. Try pure software implementation (cont'd)



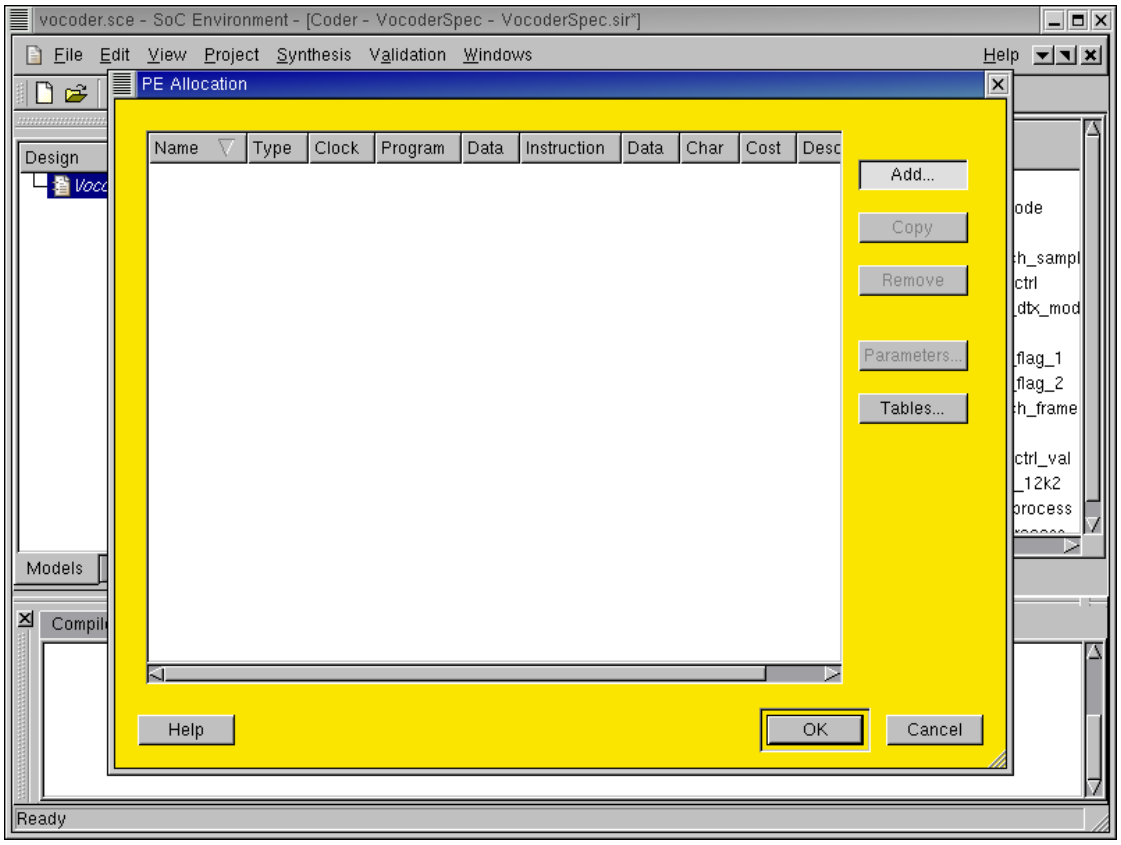
As shown in the figure, when the top level behavior "Coder" is specified, the names of all its child behaviors are italicized to distinguish them from the test bench behaviors. In general, any behavior which needs to be tested can be set as top level. So, in a generic sense, the design under test can be identified by the italicized font.

3.2.1.3. Try pure software implementation (cont'd)



We begin by exploring the available set of components in the database. This is required to select a suitable processor. To view all available components and select the desired processor, select Synthesis⟶Allocate PEs... from the menu bar.

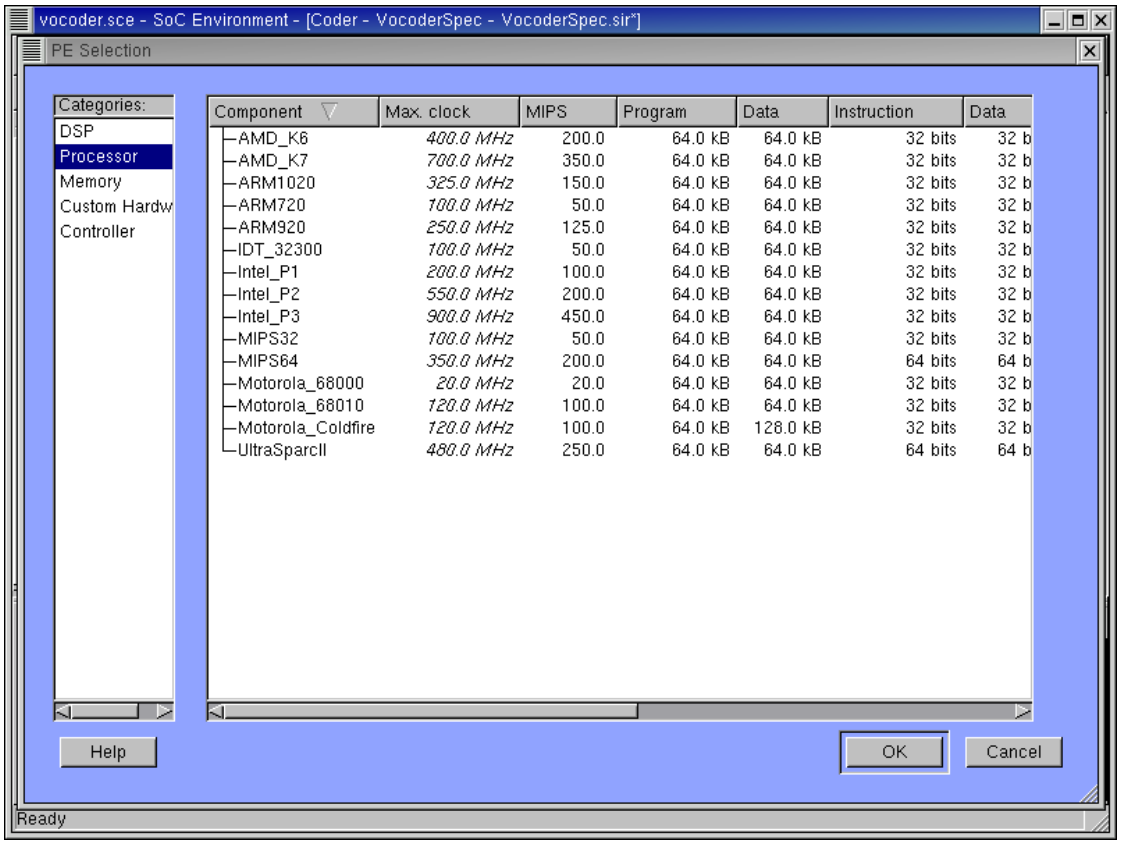
3.2.1.4. Try pure software implementation (cont'd)



Now a PE Allocation window pops up. This window includes a table to display important characteristics of components selected for the design. In addition, it also provides a number of buttons (on the right side) for user actions, such as adding a component, removing a component, and so on. Since we have not allocated any component at this point, the table has no entry.

To view the component database and select the desired component, press the Add... button.

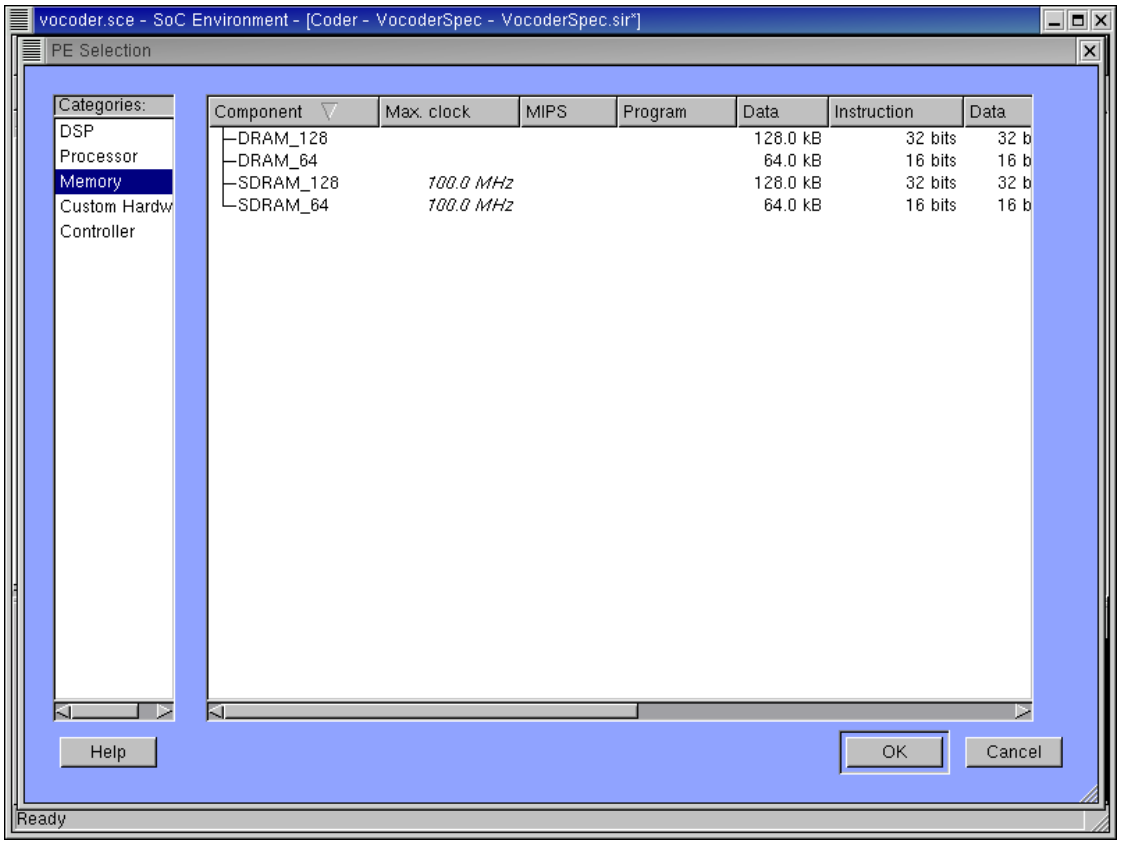
3.2.1.5. Try pure software implementation (cont'd)



Now a PE Selection window is brought up. The left side of the window (Categories) lists five categories of components stored in the database. The right side of the window displays all components within a specific category along with their characteristics. As shown in the above figure, since the Processor category is selected on the left side, 15 commonly used processor components are displayed in detail on the right side.

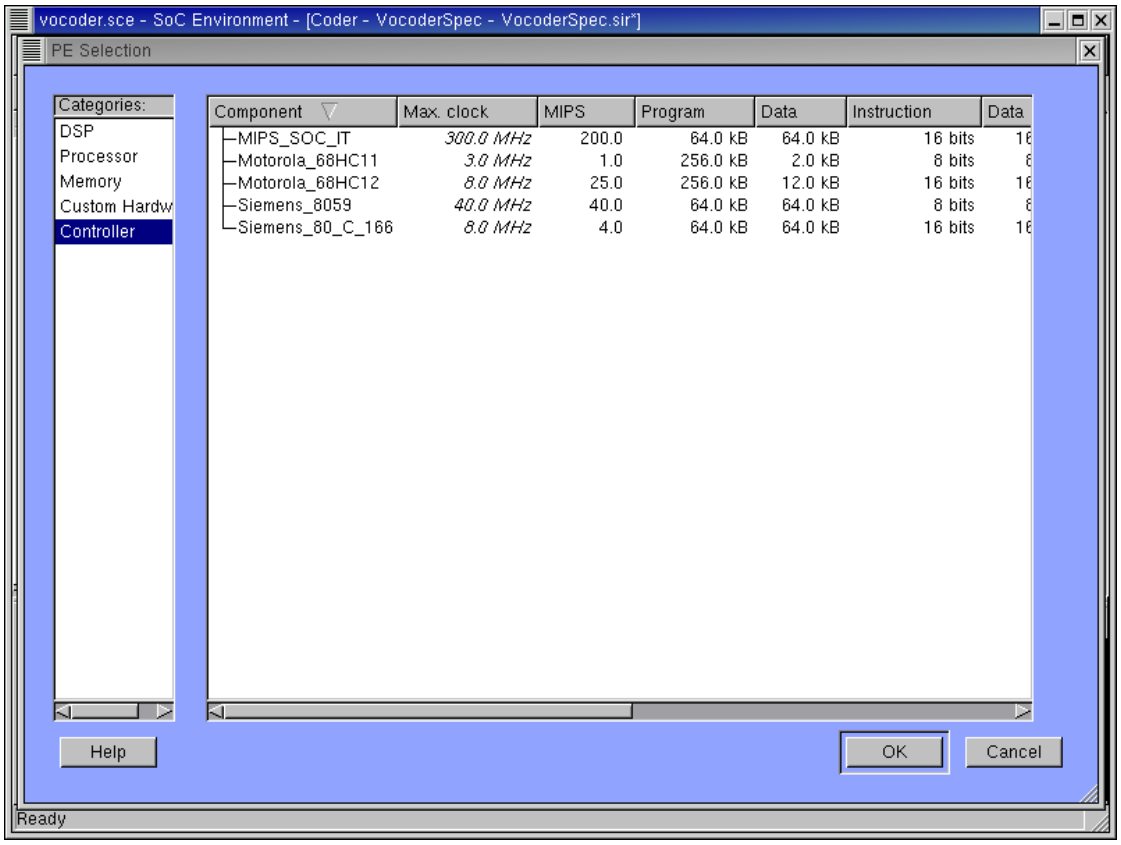
The Component description includes features like maximum clock speed, measure of the number of instructions per second, a cost metric, cache sizes, instruction and data widths and so on. These metrics are used for selecting the right component. Remember that the profiling data has given us an idea of what kind of component would be suitable for the application at hand.

3.2.1.6. Try pure software implementation (cont'd)



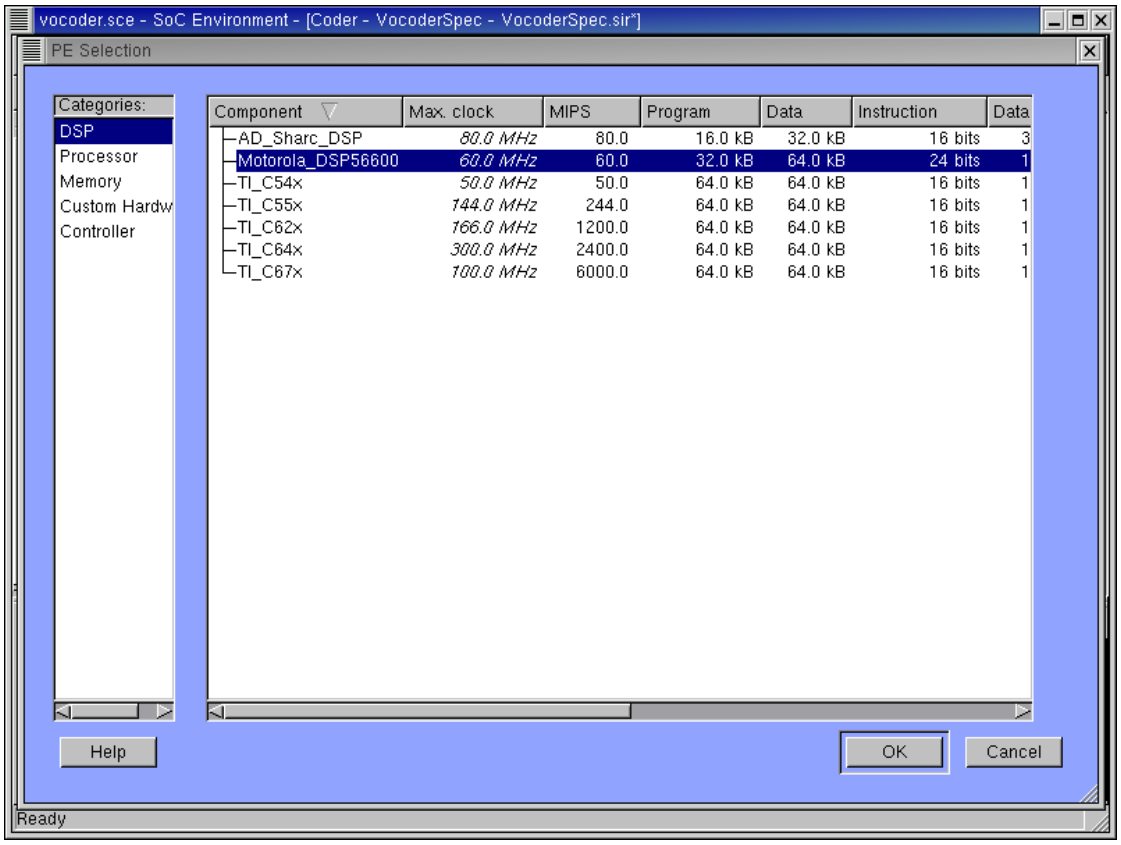
Now if we go to the Mem category, a number of memory components will be displayed in detail on the right side of the window. If the memory in the processor is insufficient for the application, we can add external memory components from this table.

3.2.1.7. Try pure software implementation (cont'd)



Now if we go to the Controller category, a number of widely used micro-controller components will be displayed in detail on the right side of the window.

3.2.1.8. Try pure software implementation (cont'd)

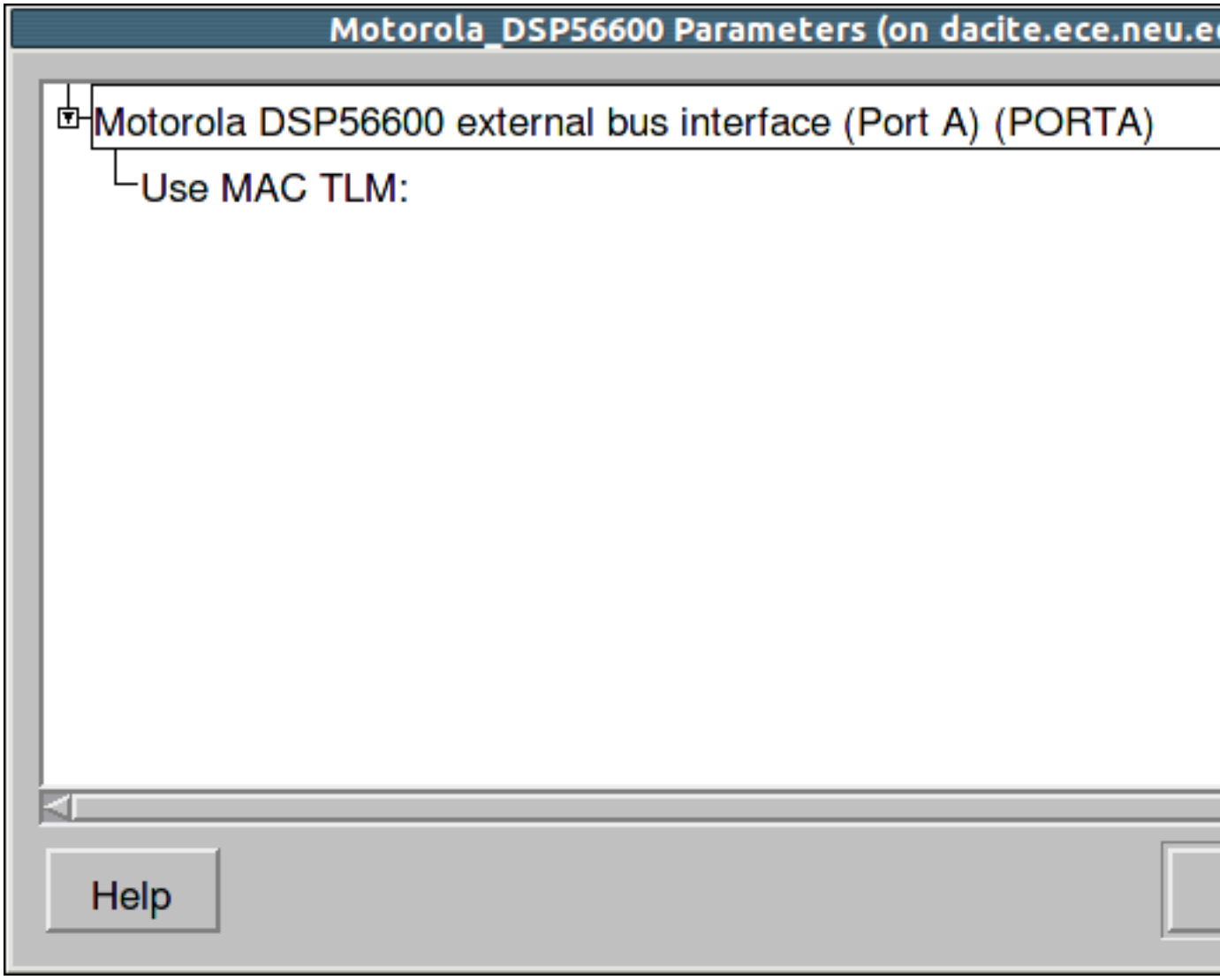


Through earlier profiling and analyzing, we found out that integer multiplication is the most significant operations in the original specification. Therefore, a fixed-point DSP would be desirable for this design.

Under the DSP category, a number of commercially available DSPs are displayed. These DSP components are maintained as part of the component library and may be imported into the design upon requirement. Since the Vocoder design project was supported by Motorola, our first choice is DSP56600 from Motorola.

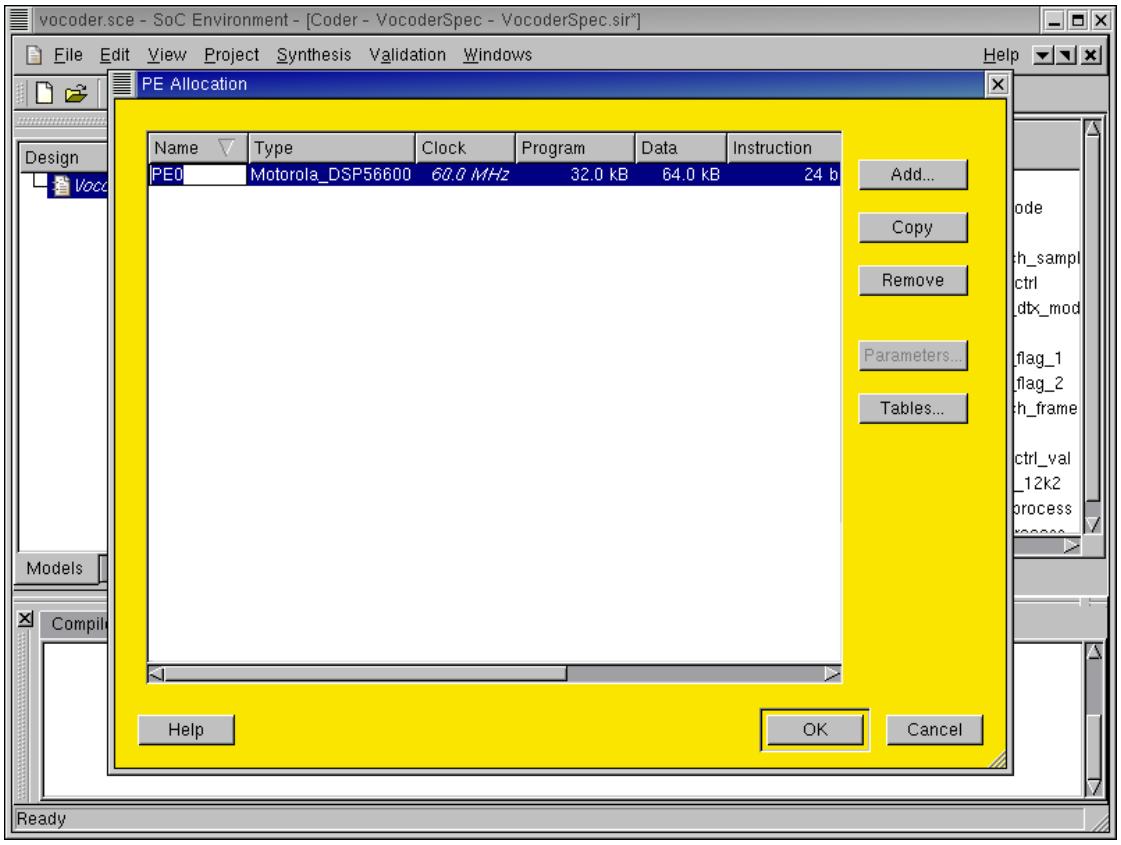
Left click the "Motorola_DSP56600" row to select it. Then click OK button to confirm the selection.

3.2.1.9. Try pure software implementation (cont'd)



After clicking OK to confirm the selection in the PE Selection dialog, a new dialog will pop up to allow entering parameters of the allocated Motorola DSP. Use the default parameters, i.e., accept the dialog by clicking OK.

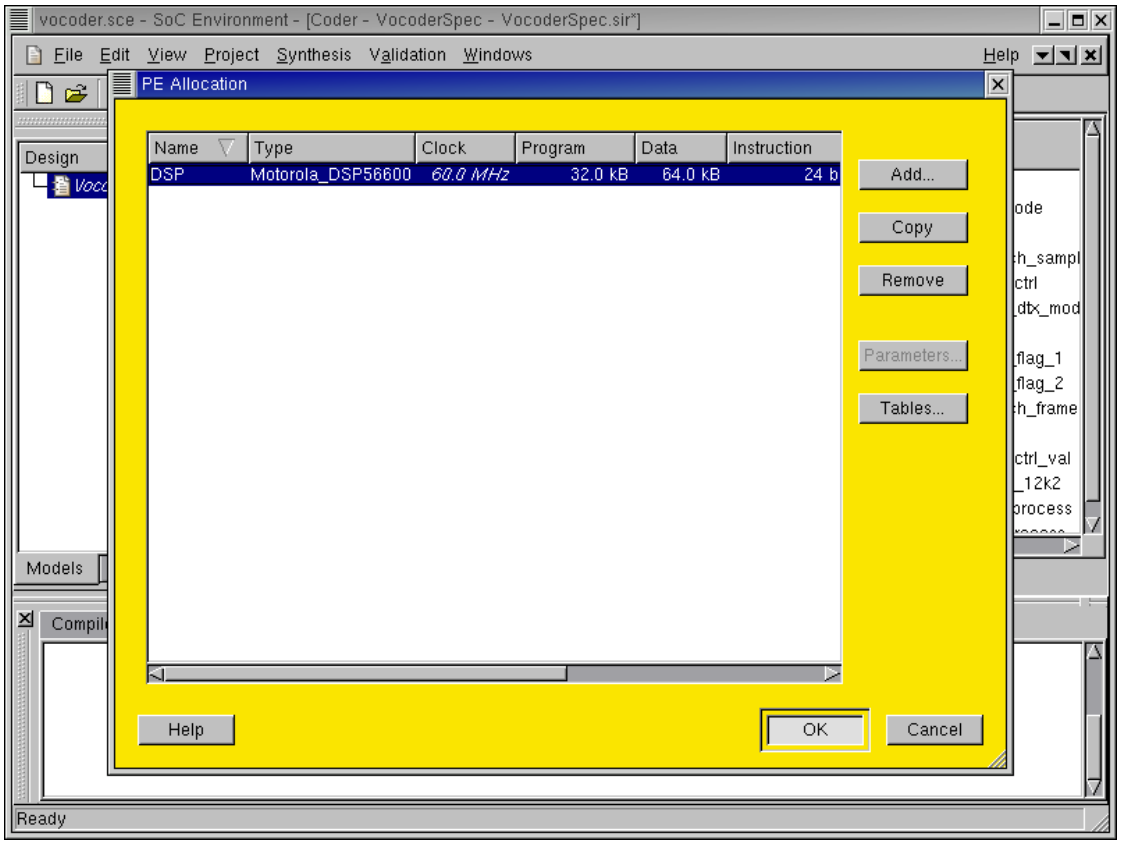
3.2.1.10. Try pure software implementation (cont'd)



Now the PE Selection window goes away and the PE Allocation table has one row that corresponds to our selected component, which has a type of "Motorola_DSP56600". This new component was named as "PE0" by default. To make it more descriptive for later reference, it is desirable to rename it.

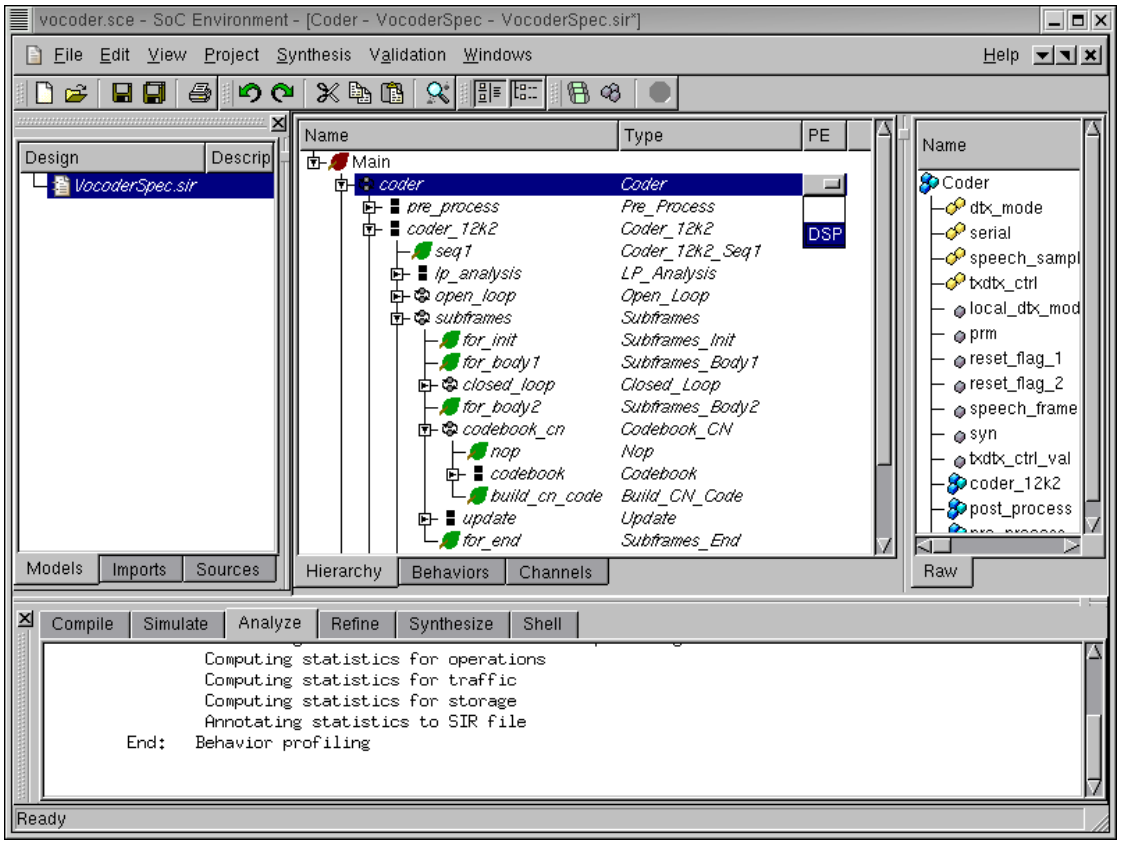
To rename it, just left click in the Name column of the row. The cursor will be blinking to indicated that the text field is ready for editing.

3.2.1.11. Try pure software implementation (cont'd)



We will simply name the component as "DSP" since it is the only component used in the design at this instance. Proceed by typing "DSP" in the text field and press return to complete the editing. Then press the OK to finish component allocation.

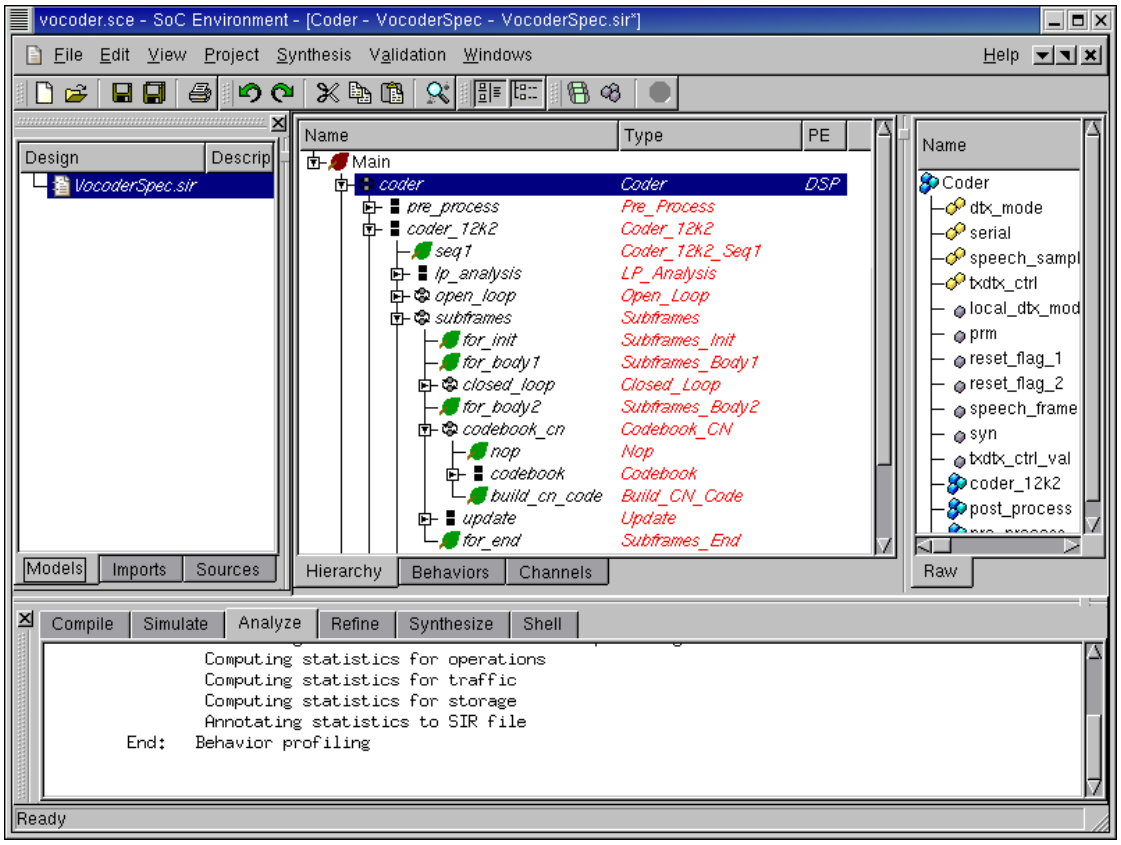
3.2.1.12. Try pure software implementation (cont'd)



As mentioned earlier, we will map the whole design to the selected processor. This is done by assign the top level behavior "Coder" to "DSP". Left click in the PE column in the row for the "Coder" behavior. A drop box containing allocated components comes up. Left click on "DSP" to map behavior "Coder" to "DSP".

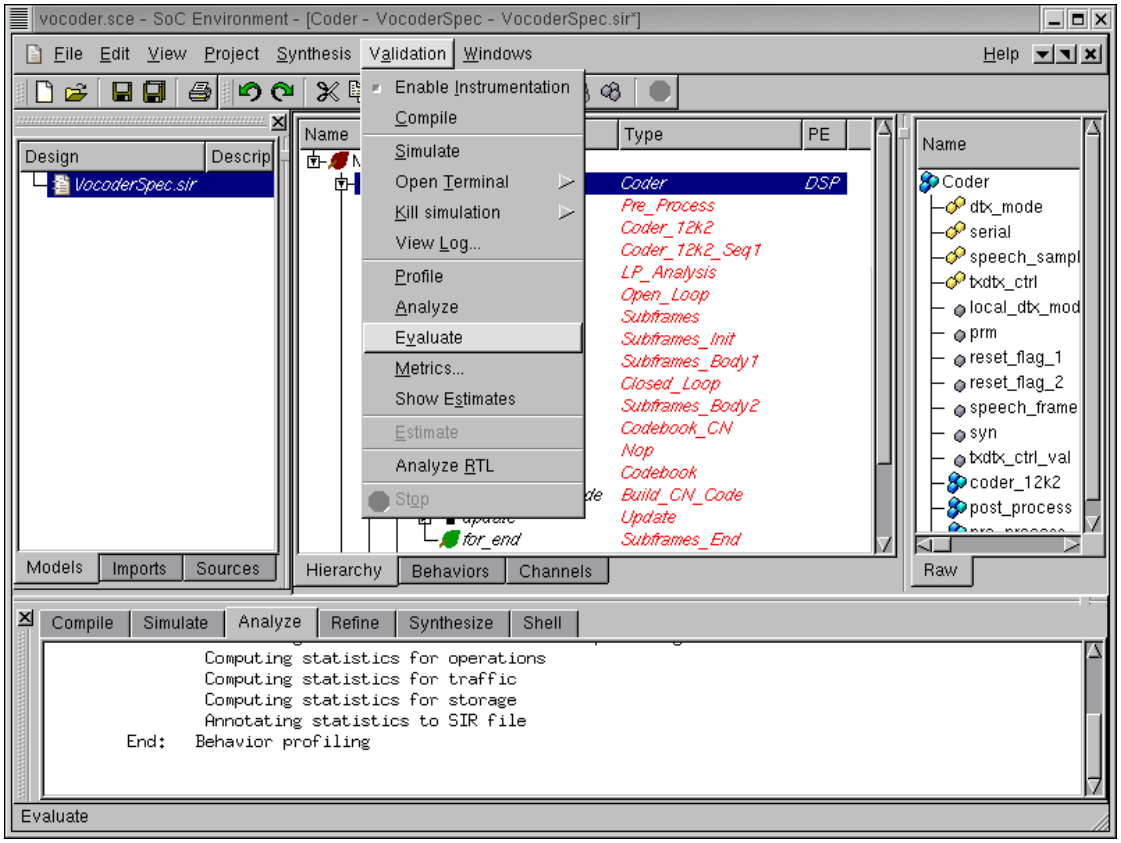
It should be noted that any kind of mapping is allowed. However, since we are investigating a purely software implementation, everything in the design gets mapped to the "DSP".

3.2.1.13. Try pure software implementation (cont'd)



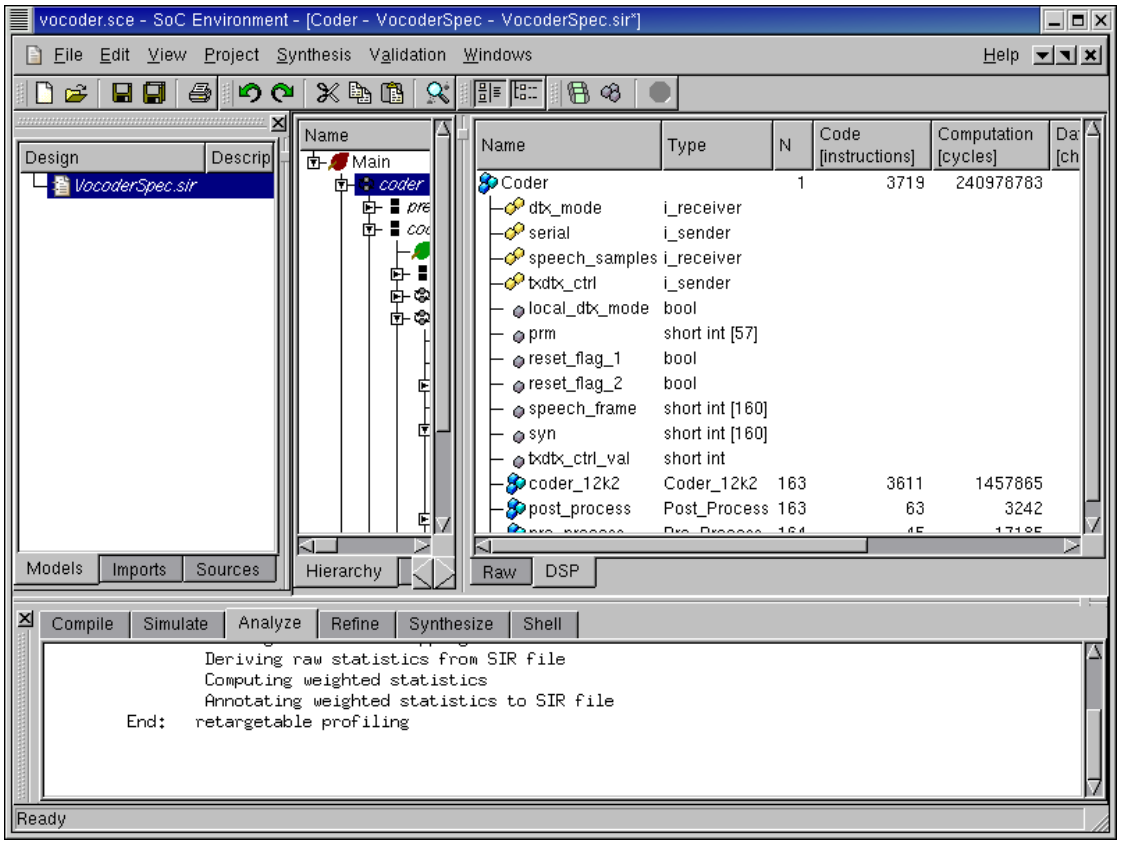
As we can see now, the descendant behaviors are all highlighted in red to indicated that they are mapped to the "DSP" component.

3.2.2. Estimate performance



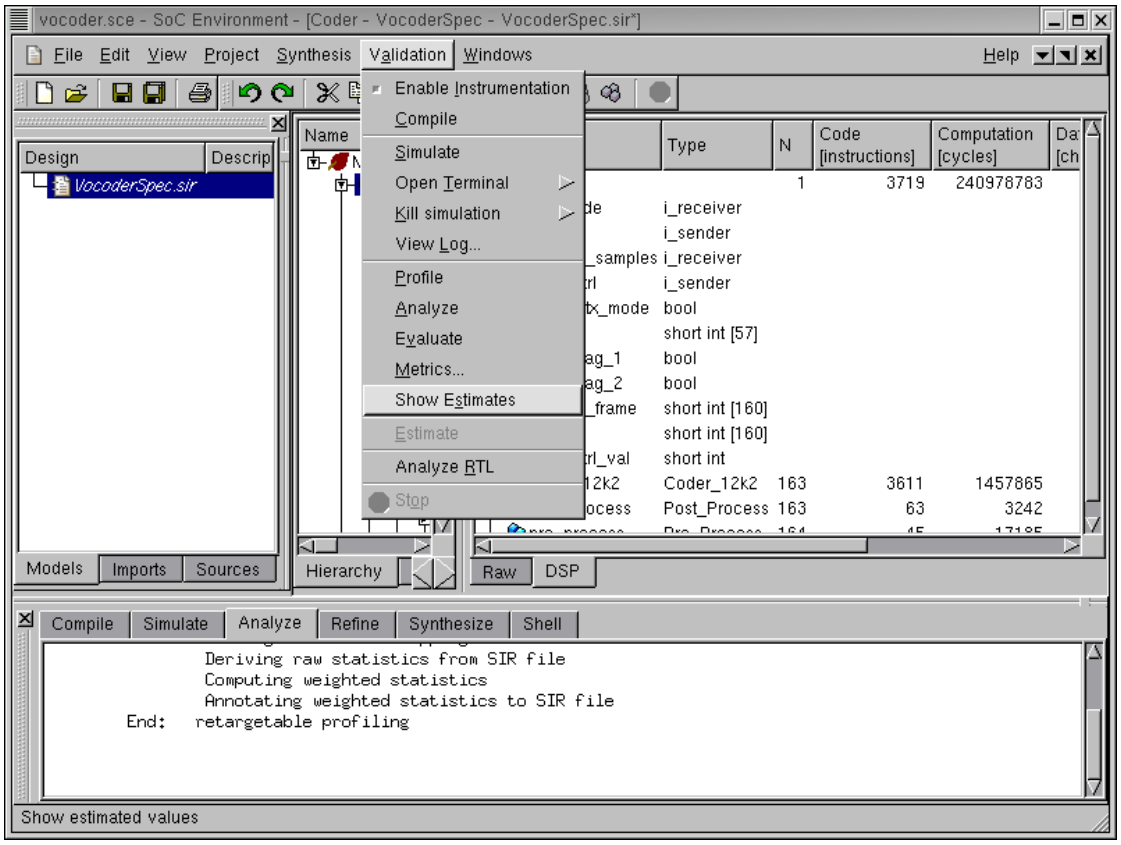
The next step is to analyze the performance of this architecture. Recall that we have a timing constraint to meet. We must therefore check if a purely software implementation would still suffice. If not, we will try some other architecture. Now we can estimate the performance of this pure software mapping by selecting Validation⟶Evaluate from the menu bar.

3.2.2.1. Estimate performance (cont'd)



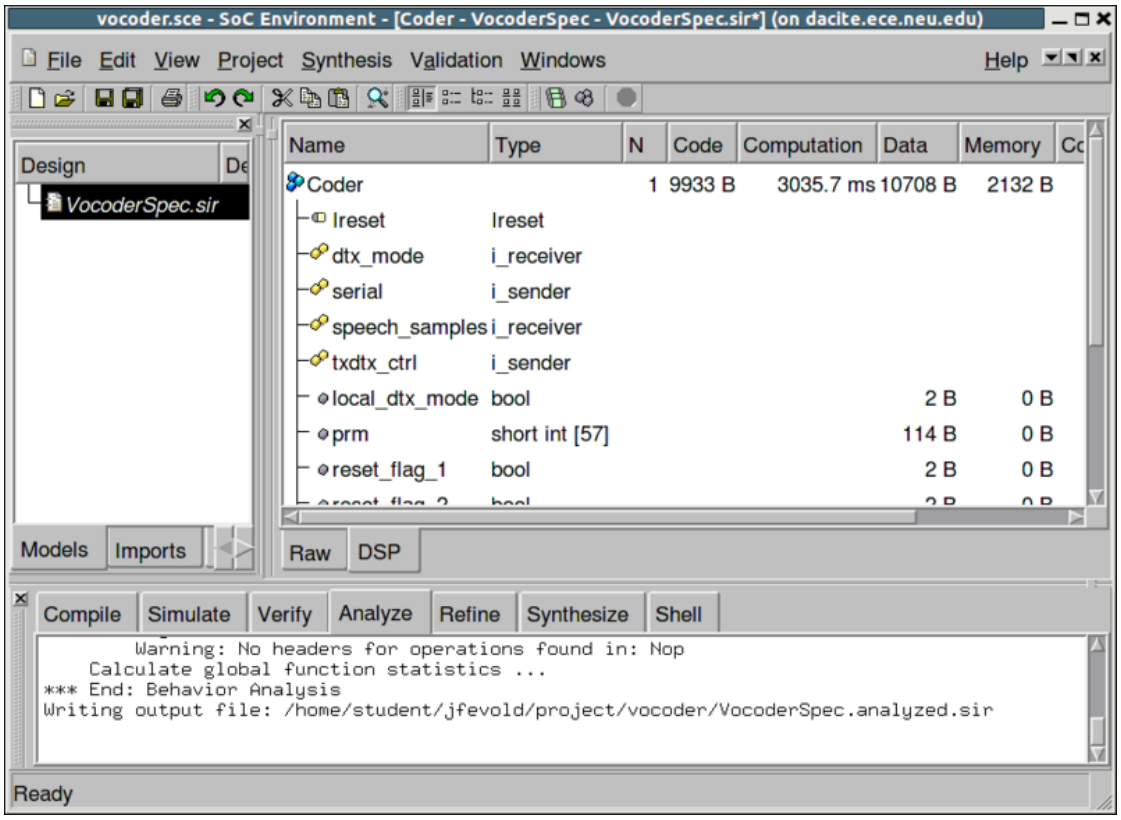
As we can see in the logging window, a re-targeted profiling is being performed. Notice in the log information that raw statistic generated during profiling are used here. The raw statistics are take as an input to the analysis tool that generates statistics for the current architecture. Since, we know the parameters of the DSP, the analysis tool can provide a more accurate measure of actual timing. When that is done, the profiled data is displayed in the design window with the "DSP" tab. Notice that this tab has appeared at the bottom of the design data. The total computation time is shown in terms of number of DSP clock cycles.

3.2.2.2. Estimate performance (cont'd)



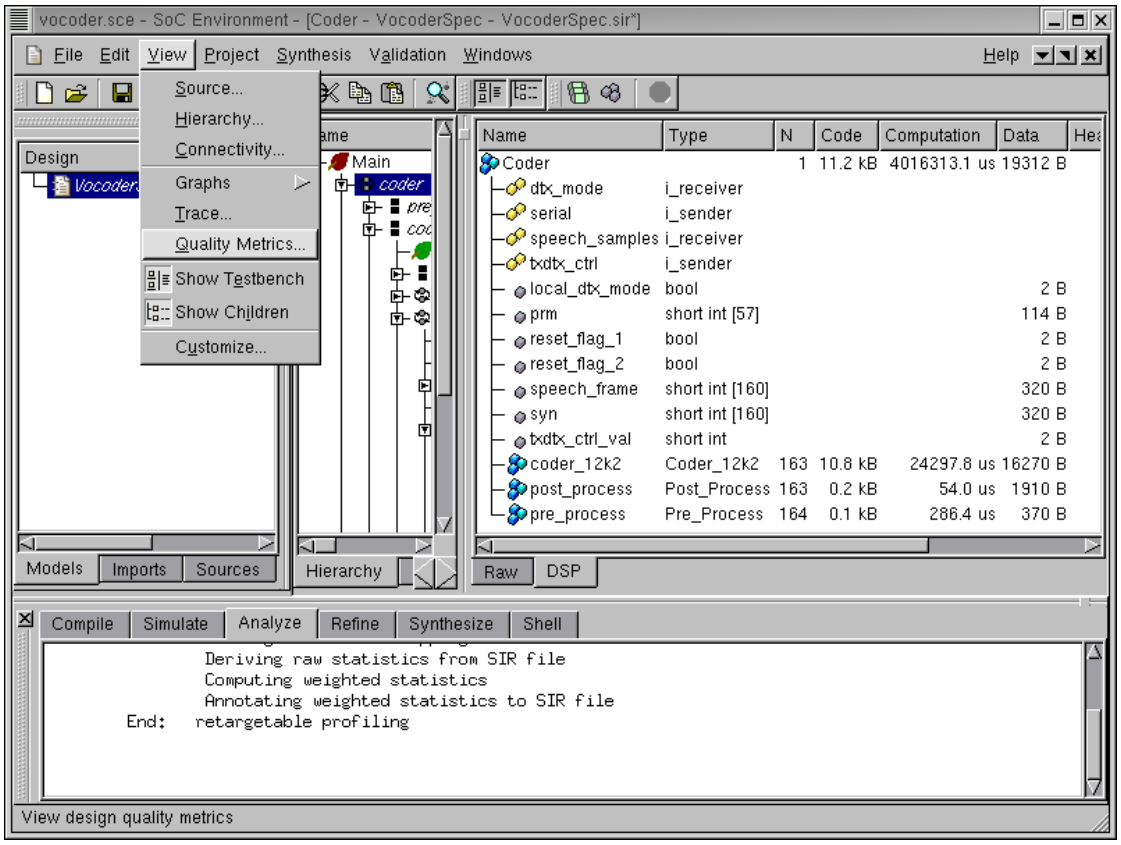
The number of computation cycles is a relevant metric for observation. However, it must be converted to an absolute measure of time so that we may directly verify if this architecture meets the demands. To find out the real execution time in terms of seconds, we turn on the option for estimation by selecting Validation⟶Show Estimates from the menu bar.

3.2.2.3. Estimate performance (cont'd)



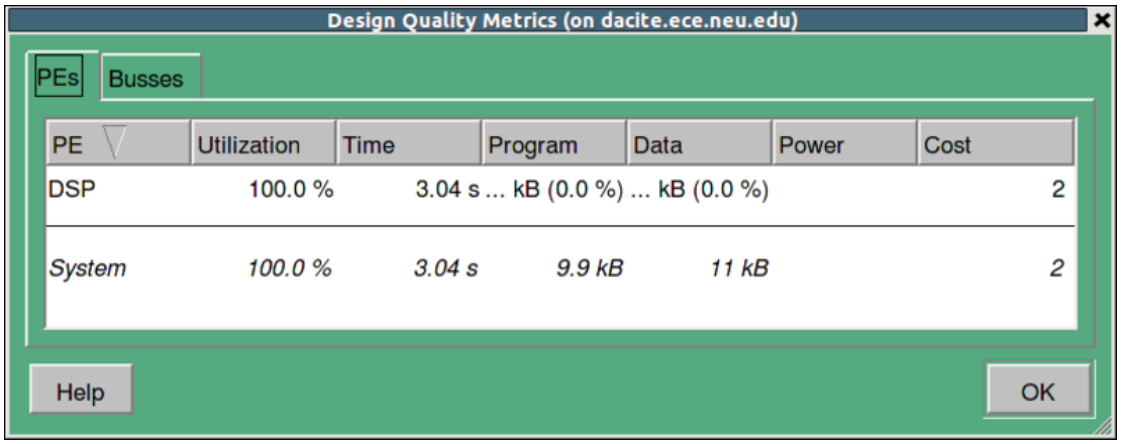
As seen in the design window, the computation time is in unit of "us". As we can see in the row of behavior "Coder", the estimated execution time (~ 4.00 seconds) exceeds the timing constraint of 3.26 seconds.

3.2.2.4. Estimate performance (cont'd)



We can also view the design quality metrics such as the execution time by selecting View⟶Quality Metrics from the menu bar.

3.2.2.5. Estimate performance (cont'd)

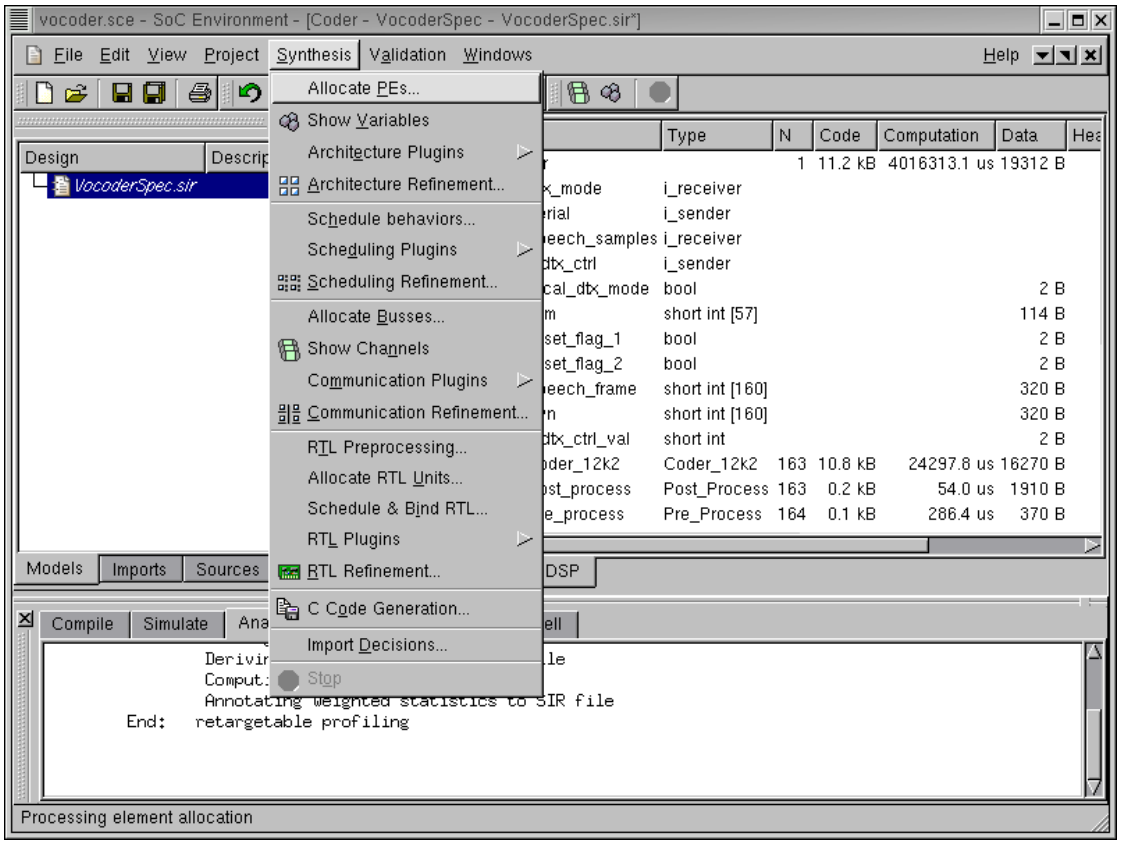


A Design Quality Metrics table pops up, showing that the estimated execution time to be 4.02 seconds, which exceeds the timing constraint of 3.26 seconds. Therefore, the pure software solution with a single "Motorola_DSP56600" does not work. We, therefore, need to experiment with other architectures. To proceed, click OK to close the Design Quality Metrics table.

3.2.3. Try software/hardware implementation

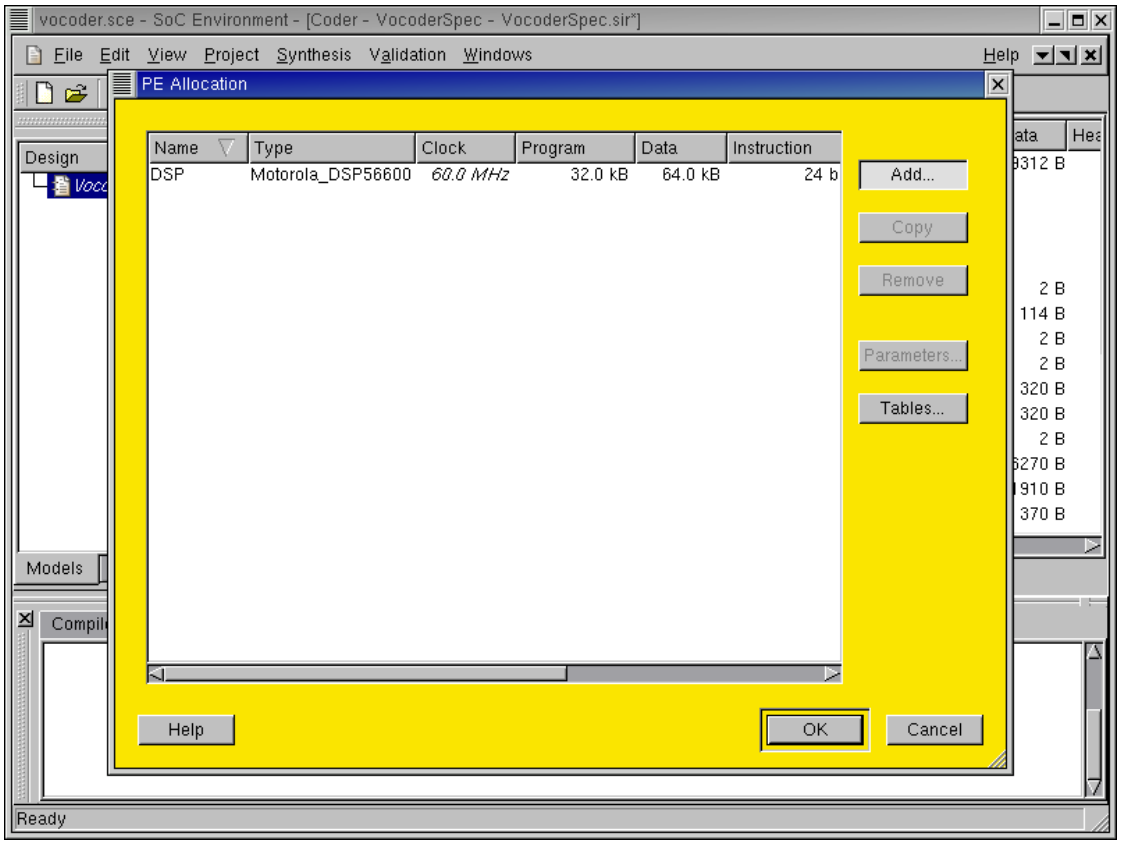
From what we observed while studying the vocoder specification, the design is mostly sequential. There is not much parallelism to exploit. What we need to reduce the execution time is a much faster component than the DSP we used. Some of the critical time consuming tasks may be mapped to a fast hardware. In this iteration, we will try to add one hardware component along with the DSP to implement the design. As we found out earlier, one of the computationally intensive and critical part in the Vocoder is the Codebook behavior. We hope to speed it up by mapping it to a custom hardware component and execute the remaining behaviors on the DSP.

3.2.3.1. Try software/hardware implementation (cont'd)



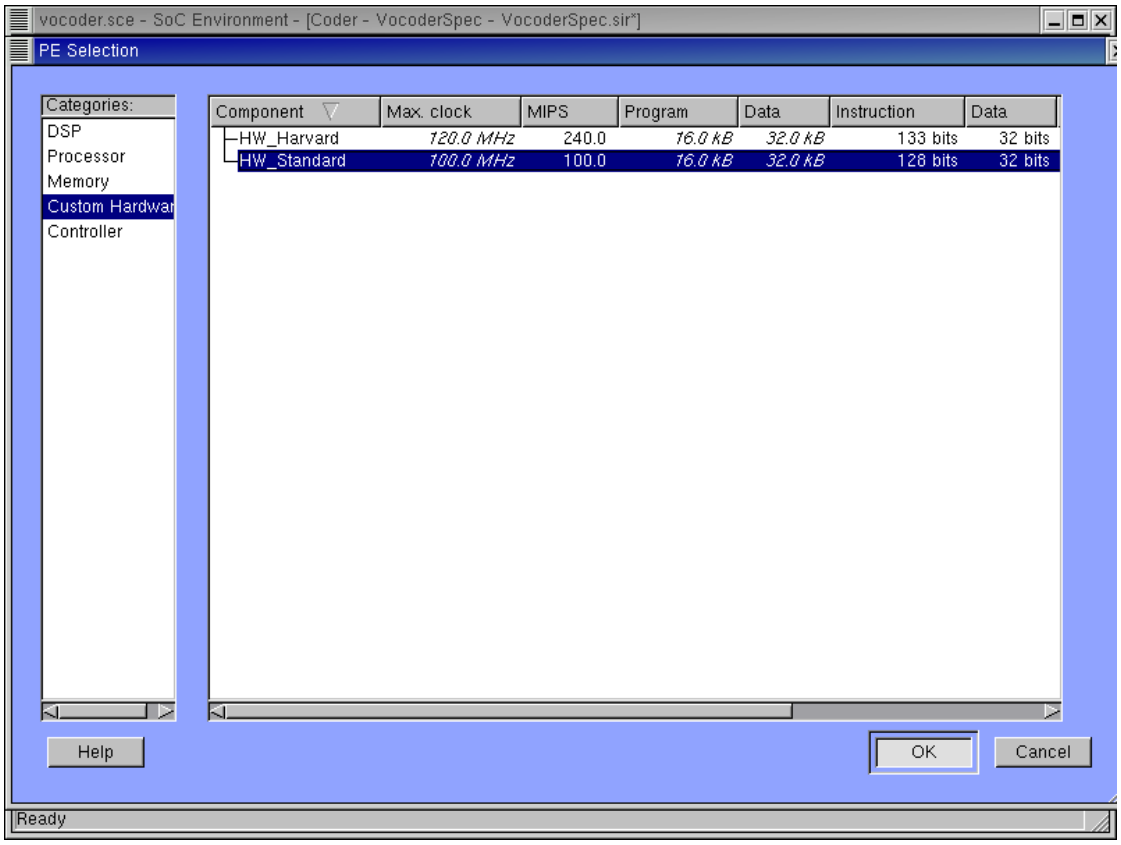
As we did earlier, while selecting the processor, go to Synthesis⟶Allocate PEs... on the menu bar.

3.2.3.2. Try software/hardware implementation (cont'd)



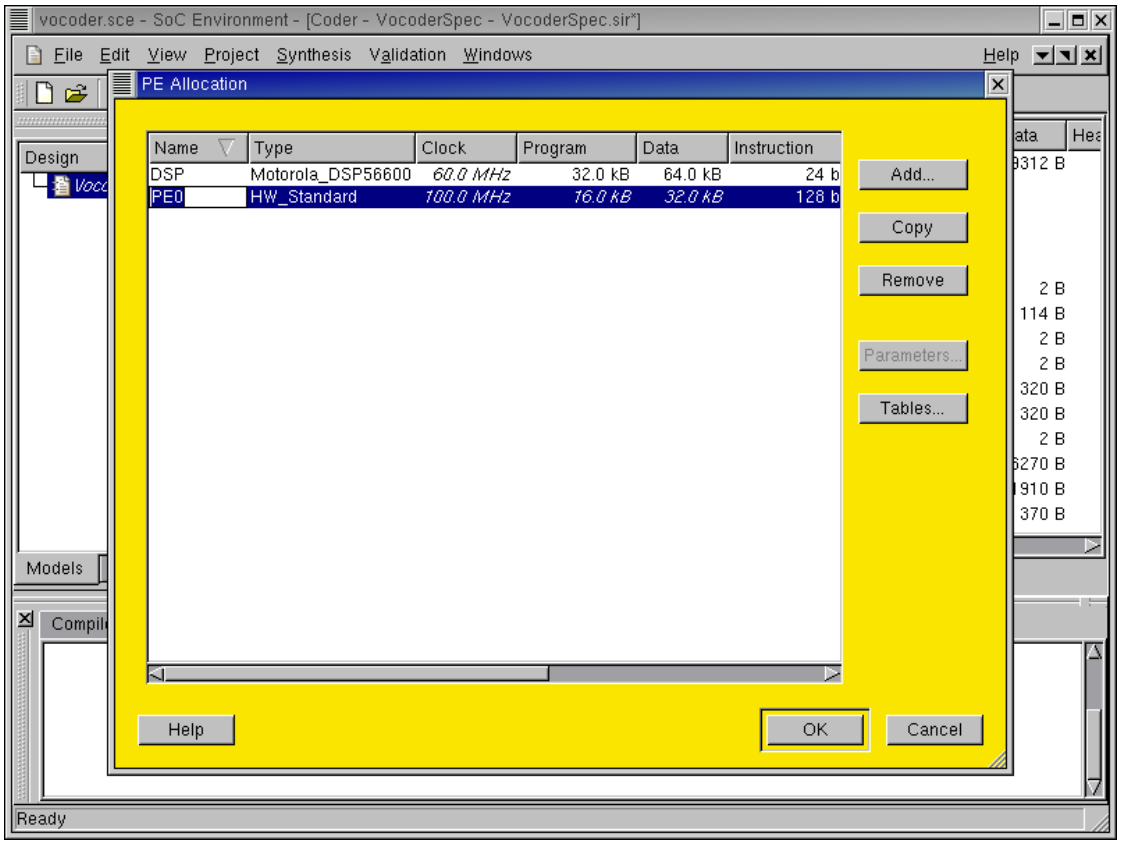
This time, the PE Allocation table pops up. As we can see, the previously allocated "DSP" component is displayed. To insert the hardware component, press Add... button to go to component database.

3.2.3.3. Try software/hardware implementation (cont'd)



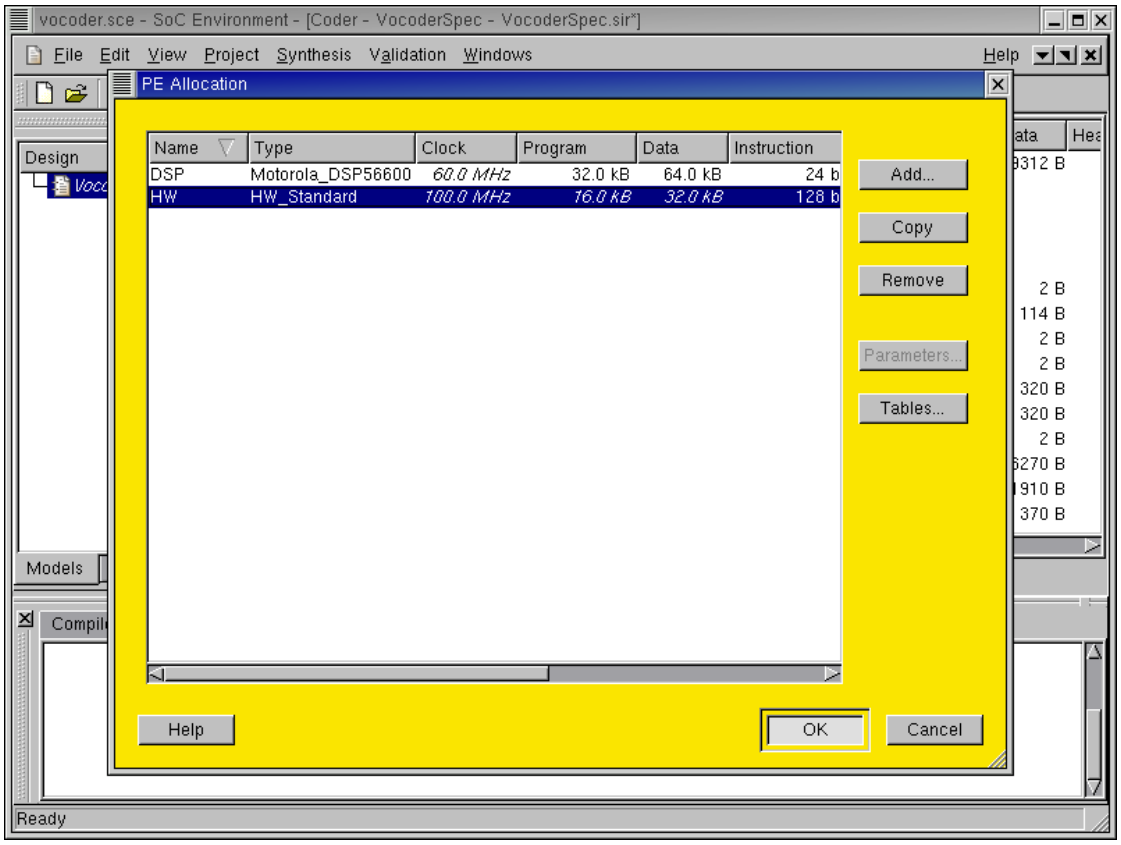
In the Custom Hardware category, two general types of hardware components are displayed. Here we will use the standard hardware design with a datapath and a control unit. Select the "HW_Standard" and press OK to confirm the selection.

3.2.3.4. Try software/hardware implementation (cont'd)



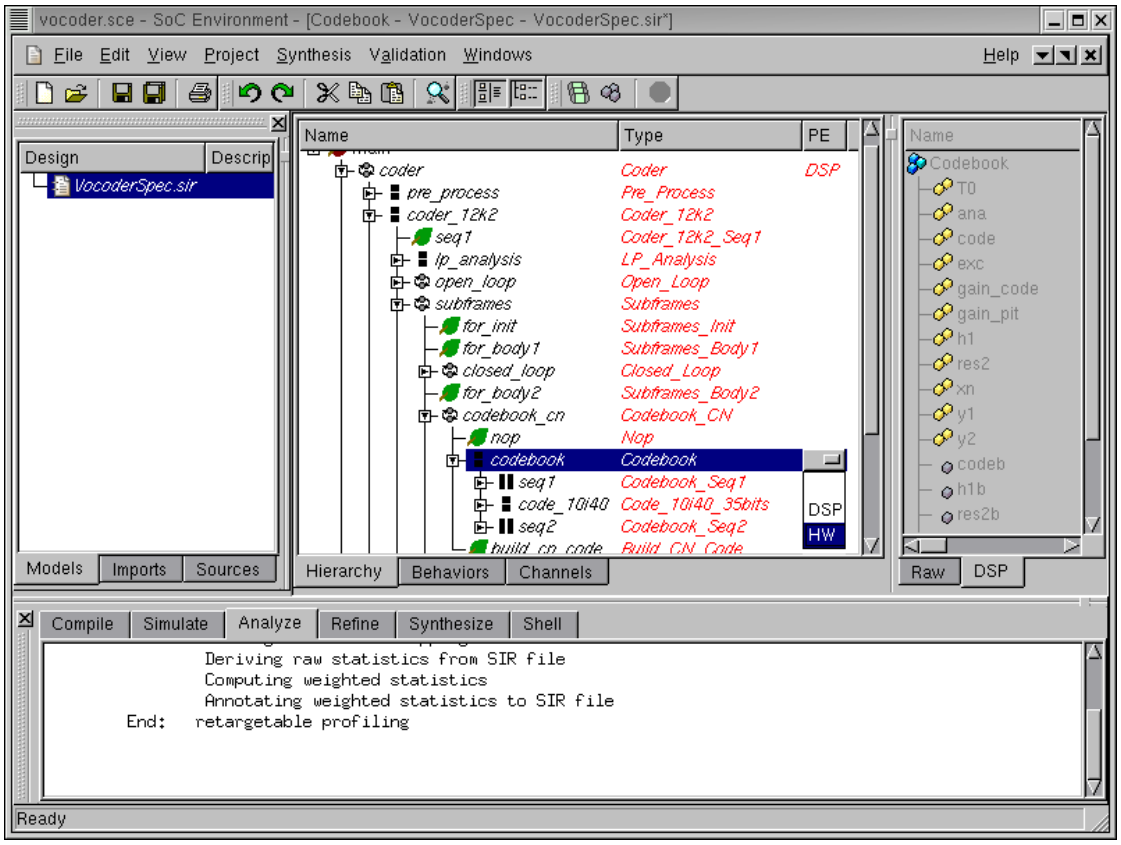
Now the "HW_Standard" component is added to the PE Allocation table. In the same way we did for the "DSP" component, we simply rename it to "HW" to distinguish it. Notice that for the hardware component, some metrics are flexible. For instance, the clock period may be changed. However, we stay with the current speed of 100 Mhz for demo purpose.

3.2.3.5. Try software/hardware implementation (cont'd)



After we renamed it, press OK button to complete component allocation.

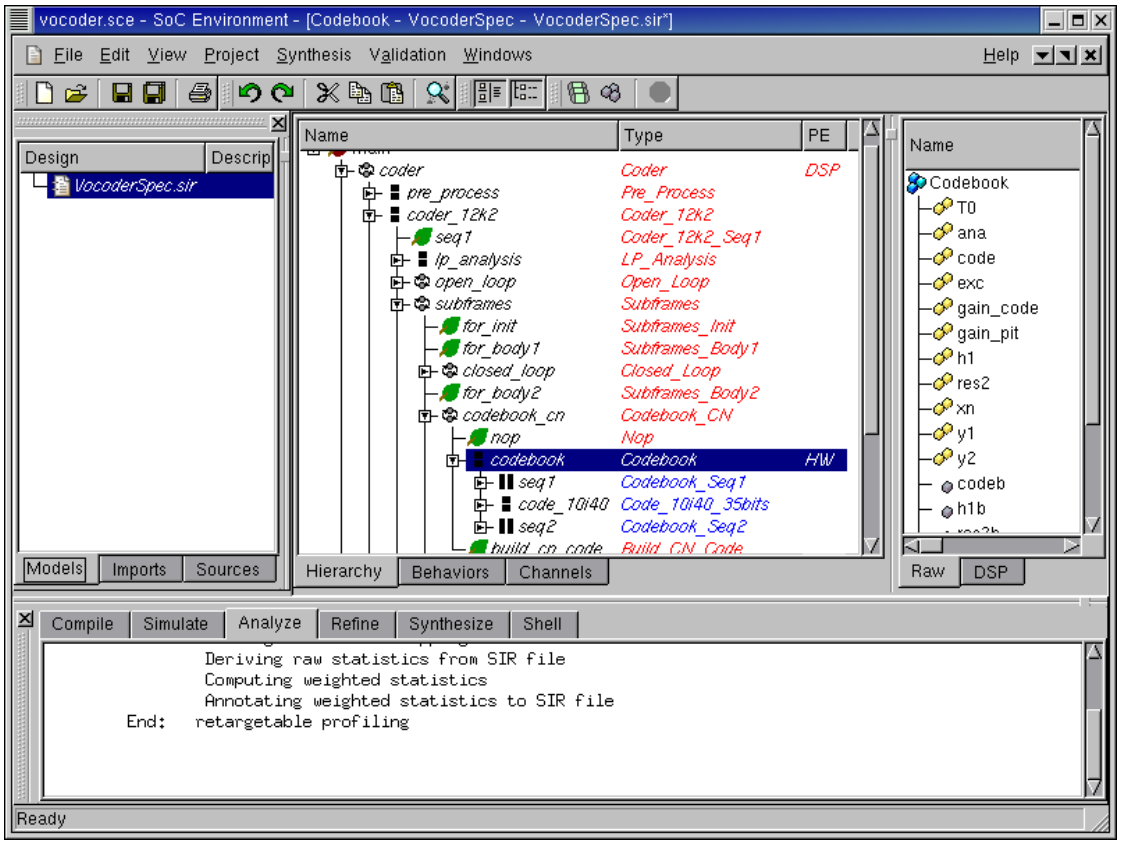
3.2.3.6. Try software/hardware implementation (cont'd)



Remember we have already specified the top level behavior and mapped all behaviors to "DSP" in the first iteration. That information is still there and we do not have to specify it again. We only need to map behavior "Codebook" to the "HW" component, as suggested earlier.

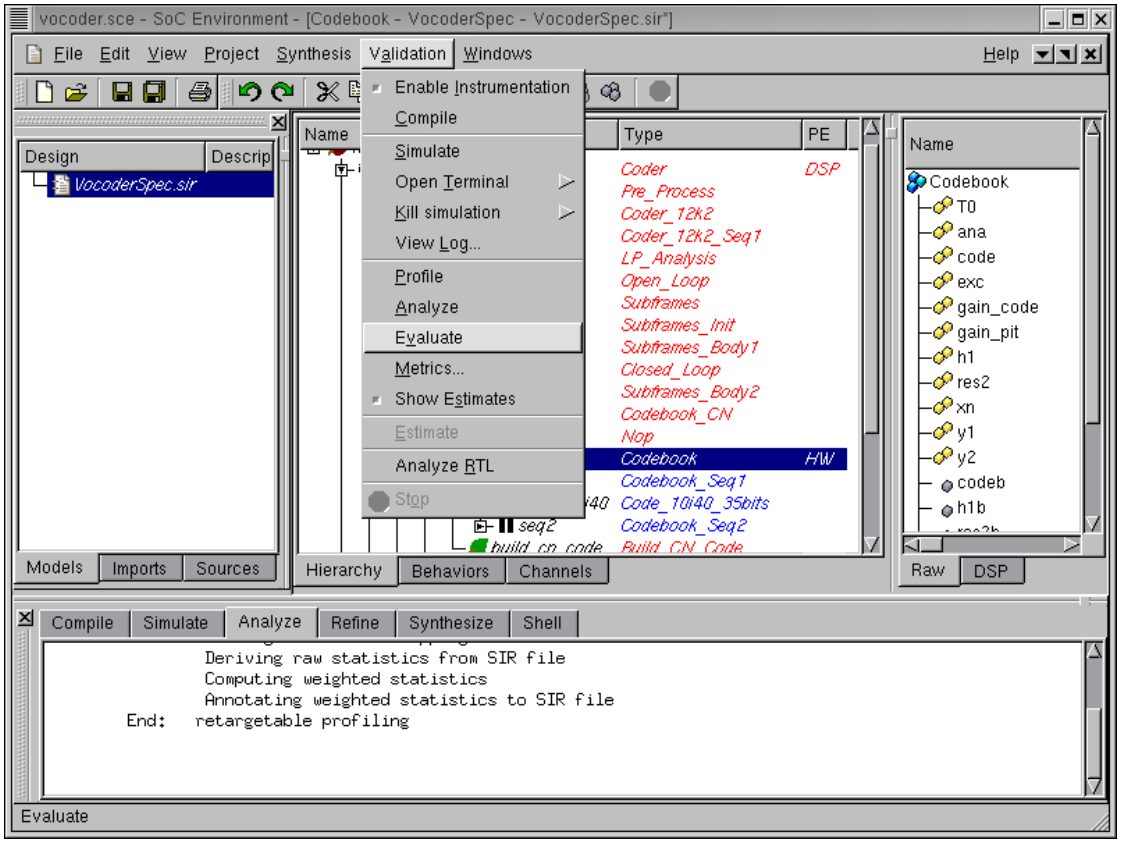
Browse the hierarchy tree to locate behavior "Codebook". Click on "Codebook" in the PE column. Click on "HW" in the drop box to map "Codebook" to "HW". This would map the entire subtree of behaviors under "Codebook" to custom hardware.

3.2.3.7. Try software/hardware implementation (cont'd)



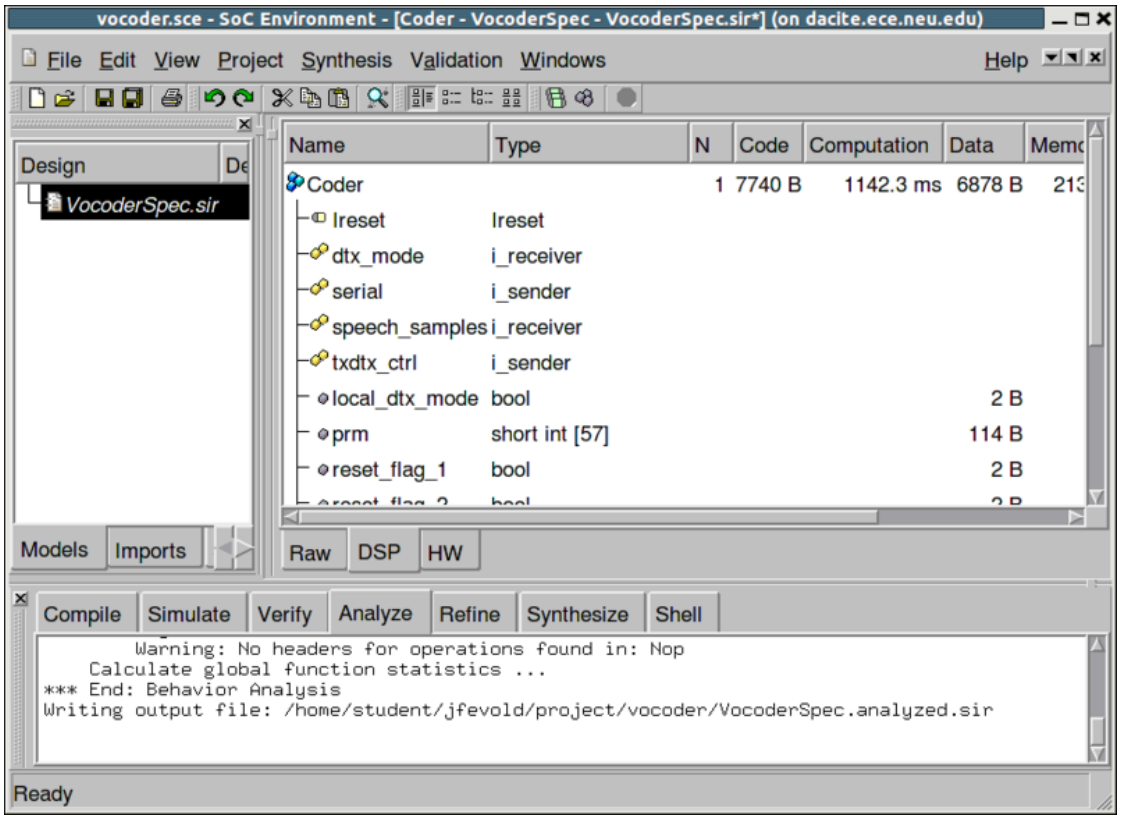
After the mapping, we will see the subtree rooted at "Codebook" is highlighted in blue in contrast to the rest behaviors in red that are mapped to "DSP".

3.2.4. Estimate performance



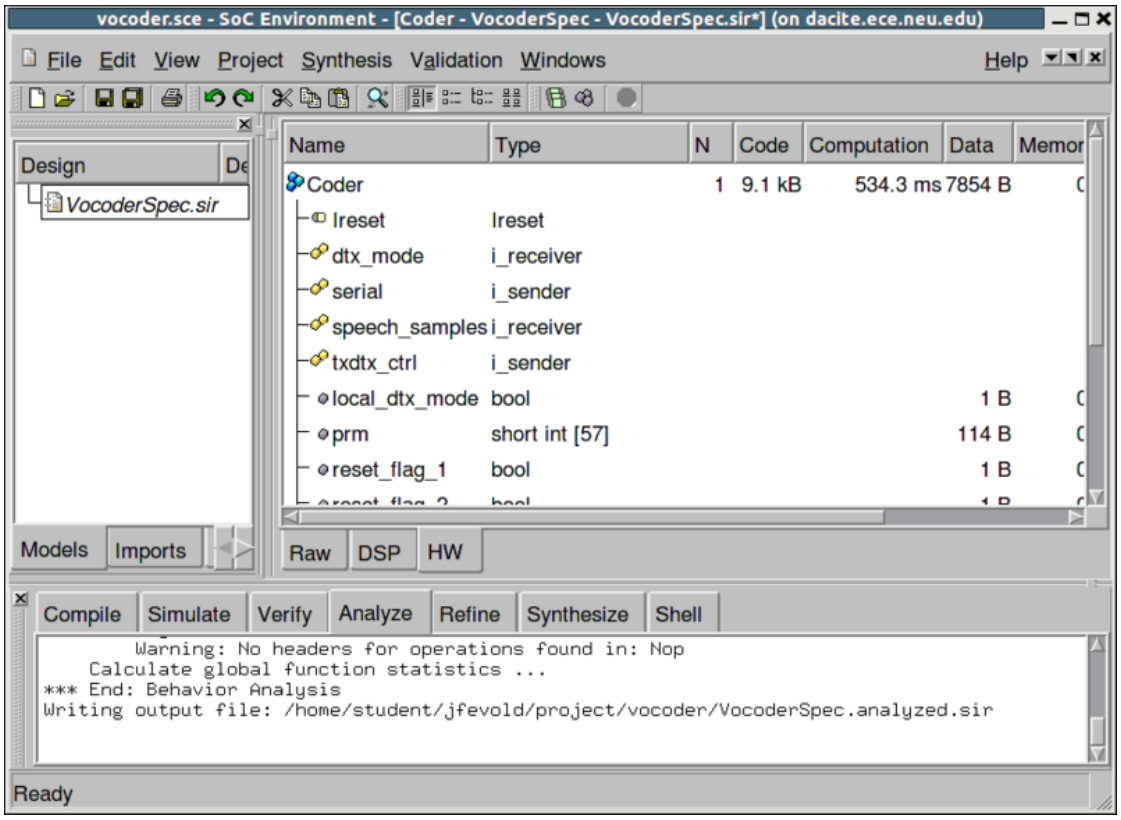
It may be recalled that we abandoned the pure software implementation because it failed on meeting the timing constraint. It is now time for us to verify if the timing is met by using the combined software/hardware design. To evaluate this software and hardware implementation, go to Validation⟶Evaluate on the menu bar.

3.2.4.1. Estimate performance (cont'd)



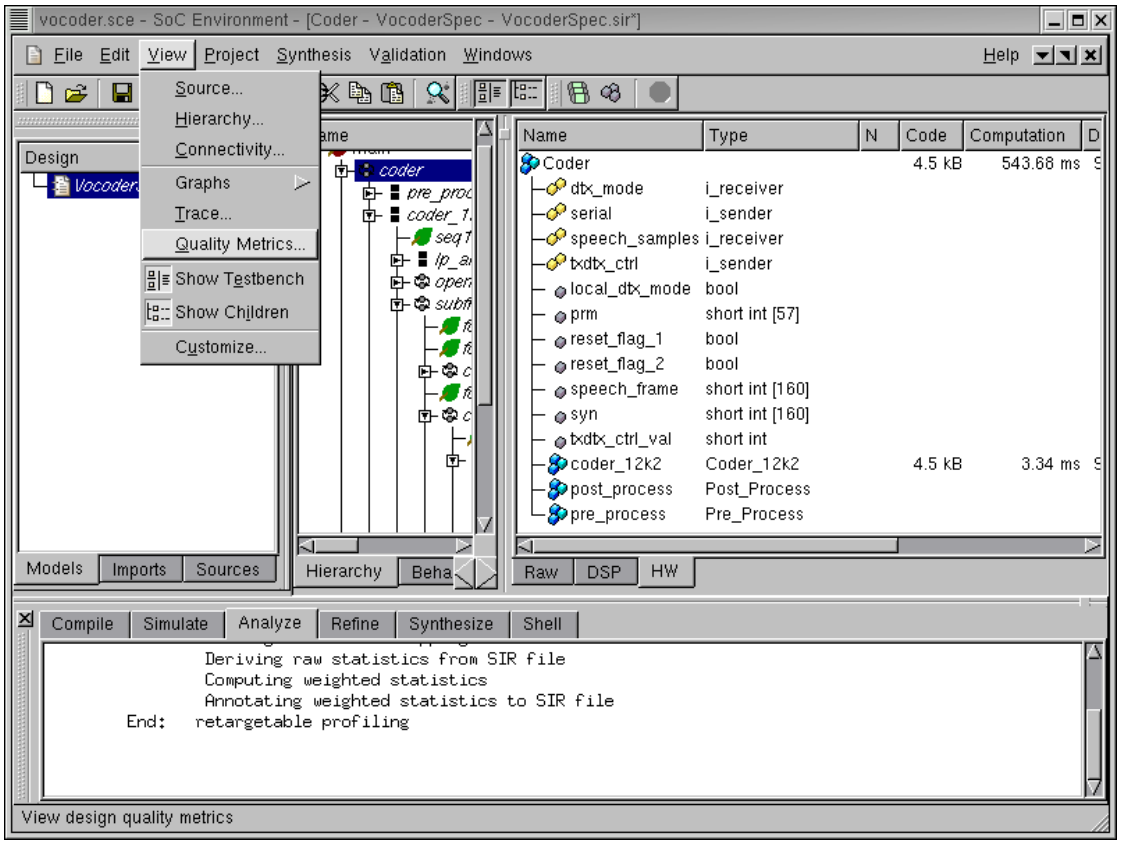
As we can see in the logging window, a profiling re-targeted at the DSP and HW architecture is being performed. When it finishes, the profiled data is presented in the design window. In order to find out the execution time of the Coder, select Coder behavior in the hierarchy tree. By clicking on the DSP tab of the view-pane, information of the DSP part of "Coder" behavior is displayed. For example, the execution time of the software part on DSP is around 1.14 seconds.

3.2.4.2. Estimate performance (cont'd)



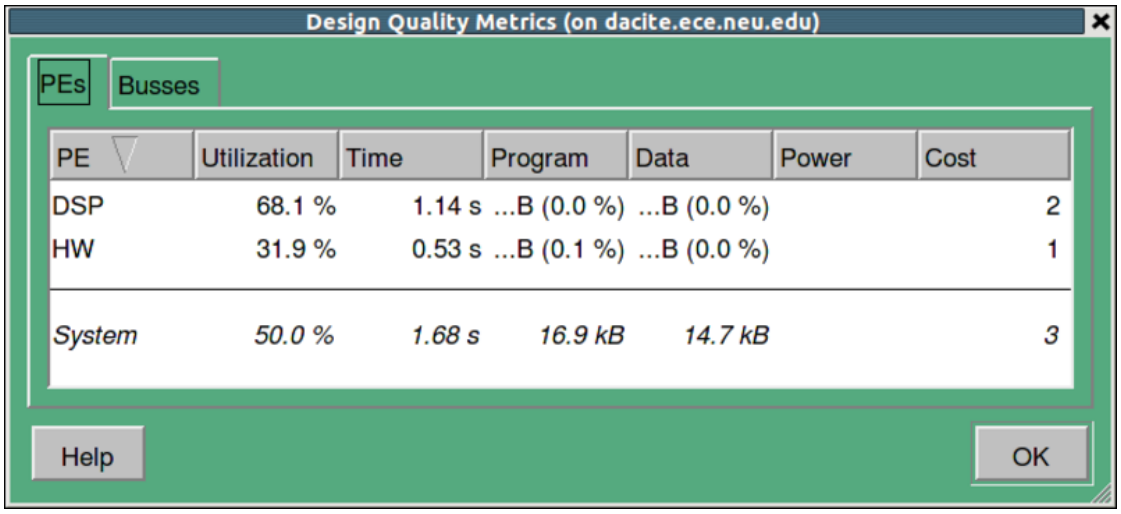
To find out the information on hardware side, click the HW tab. The view-pane shows that the execution of hardware part, behavior "Codebook", takes 0.54 seconds. Since "Codebook" was executed in sequential composition with the rest of the design, the latency of the design is the sum of DSP and HW execution time, which is 1.68 (1.14 + 0.54) seconds. Recall that the timing requirement is to be less than 3.26 seconds for the given speech data. Therefore, the current architecture and mapping are acceptable.

3.2.4.3. Estimate performance (cont'd)



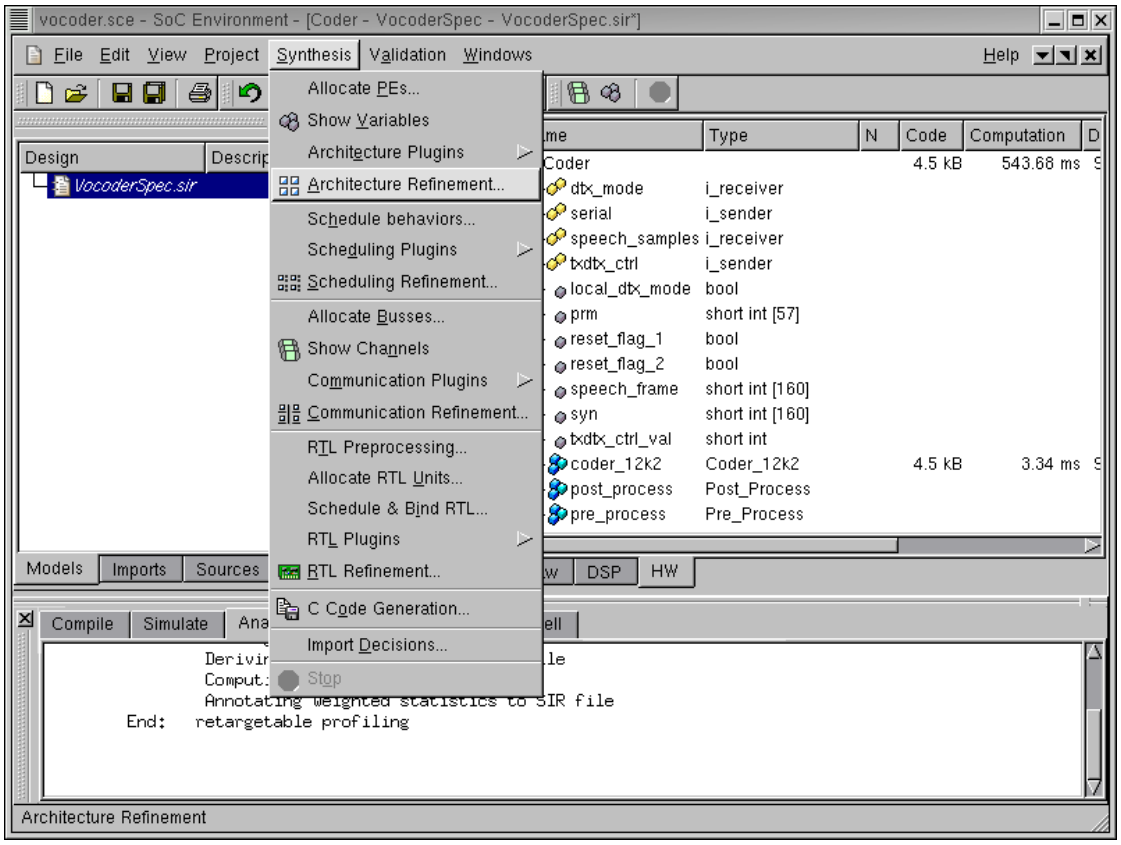
Like we did earlier, we can also view the execution time in the Design Quality Metrics table. To do so, select View⟶Quality Metrics from the menu bar.

3.2.4.4. Estimate performance (cont'd)



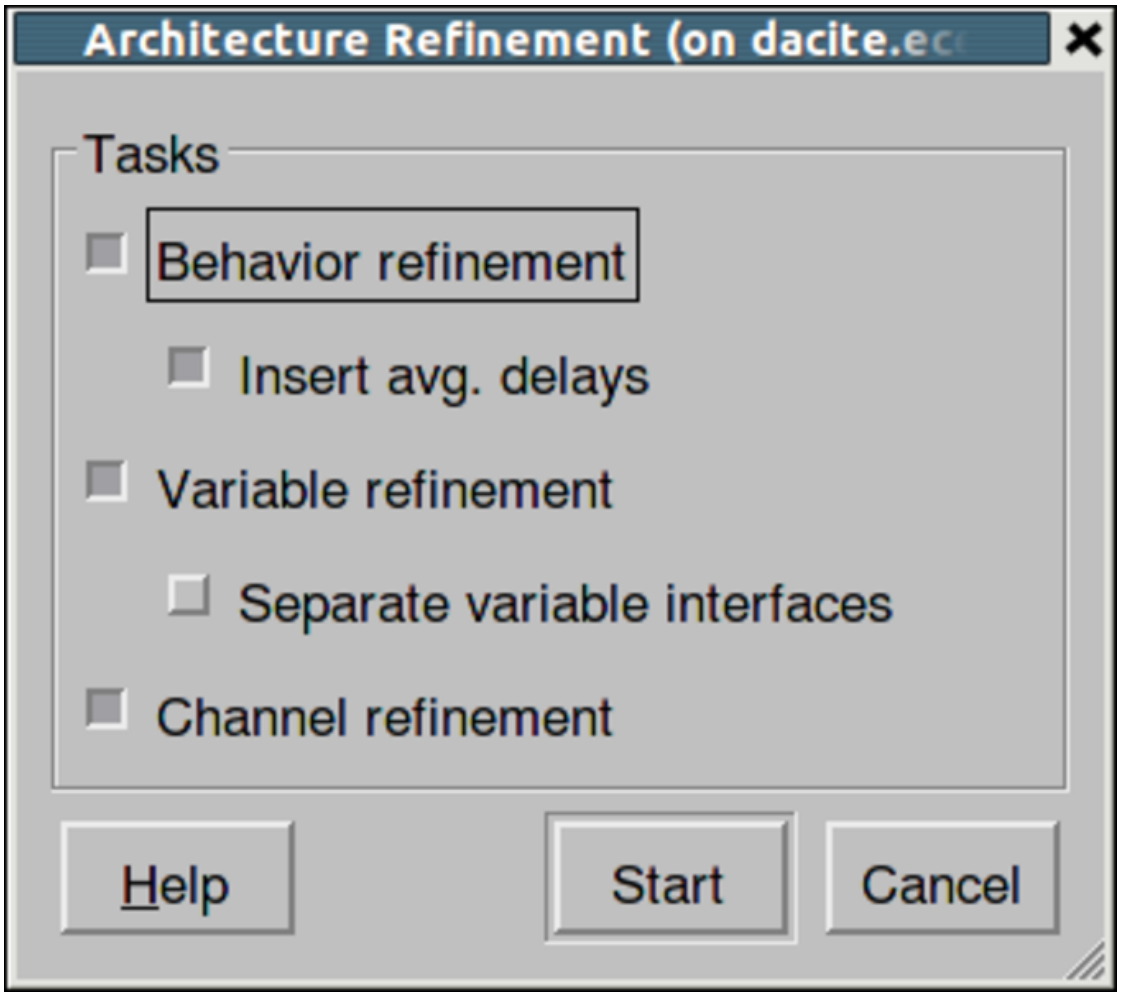
As shown in the figure, the Design Quality Metrics table including a number of design quality metrics is displayed. It confirms that the total execution time is 1.68 seconds, same as what we figured out earlier. After reviewing the quality metrics, click on OK to close the table.

3.2.5. Generate architecture model



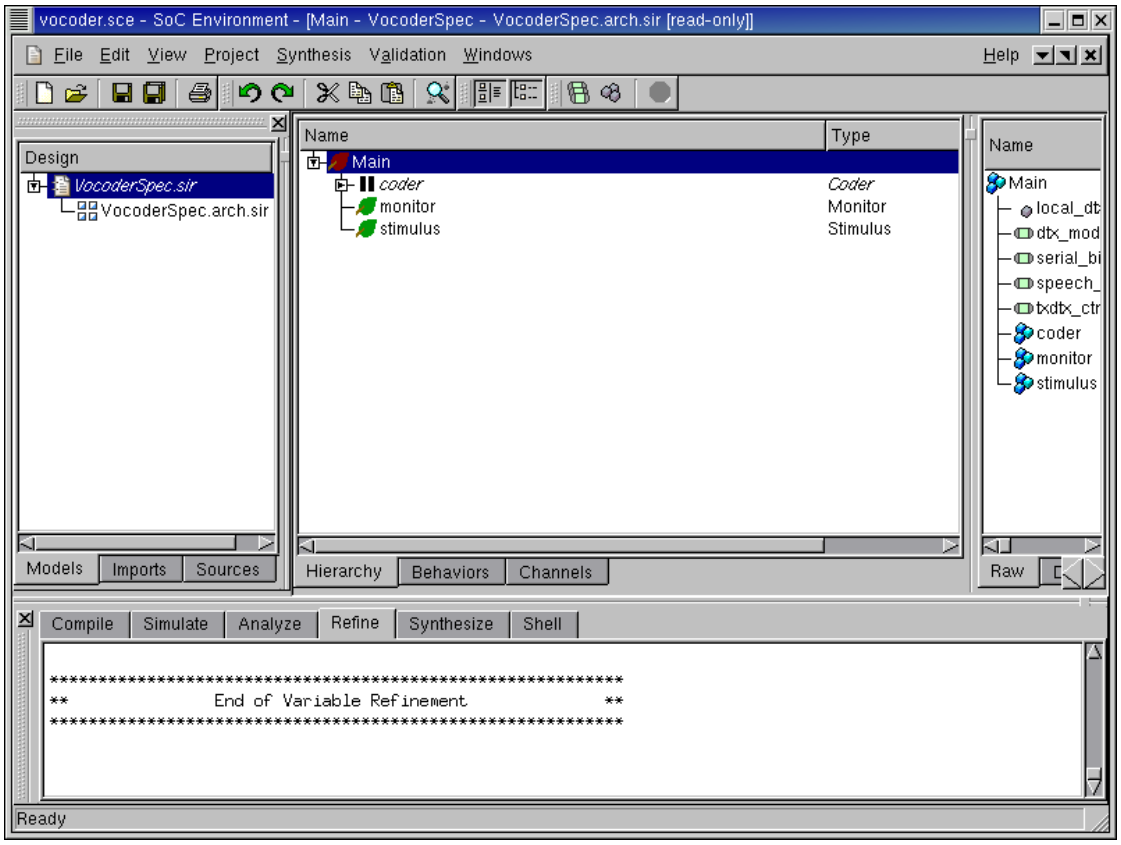
Now we can refine the specification model into an architecture model, which will exactly reflect the this architecture and mapping decisions. This can be done either manually or automatically. As we mentioned earlier, an architecture refinement tool is integrated in SCE. To invoke the tool, go to Synthesis⟶Architecture Refinement.... The tool changes the model to reflect the partition we created and also introduces synchronization between the parallely executing components. Note that we have not decided to map variables explicitly to components. For demo purposes, we will leave this decision to be made automatically by the refinement tool. However, it needs to be mentioned that the designer may choose to map variables in the design as deemed suitable.

3.2.5.1. Generate architecture model (cont'd)



A dialog box pops up for selecting specific refinement tasks of architecture refinement. By default, all tasks will be performed in one go. Now press the Start button to start the refinement. It must be noted that the user has an option to do the architecture refinements one step at a time. For instance, a designer may want to stop at behavior refinement if he is not primarily concerned about observing the memory requirements or the schedule on each component. Nevertheless, in our demo we perform all steps to generate the final architecture model.

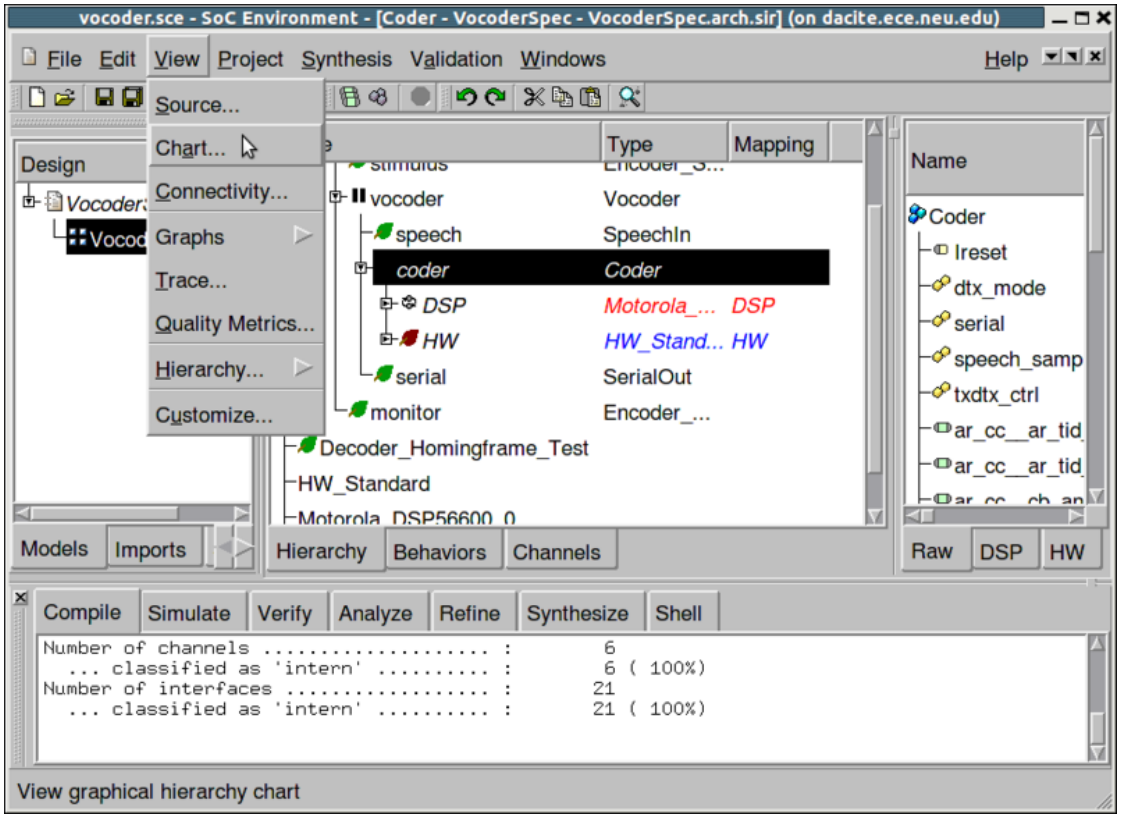
3.2.5.2. Generate architecture model (cont'd)



As displayed in the logging window, the architecture refinement is being performed. After the refinement, the newly generated architecture model "VocoderSpec.arch.sir" is displayed to the design window. It is also added to the current project window, under the specification model "VocoderSpec.sir" to indicate that it was derived from "VocoderSpec.sir". Please note that, while the architecture refinement only took a few seconds to generate, a whole new model has been created.

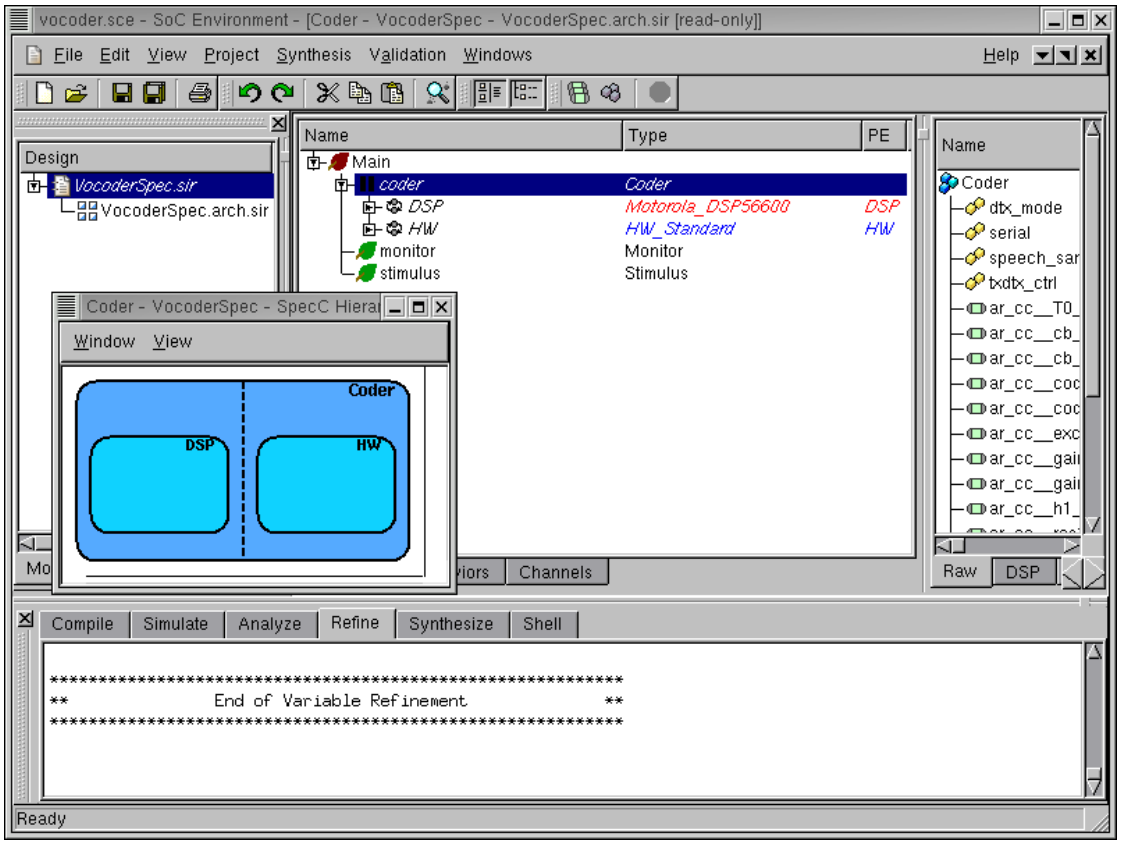
3.2.6. Browse architecture model

In this section we will look at the architecture model to see some of its characteristics.



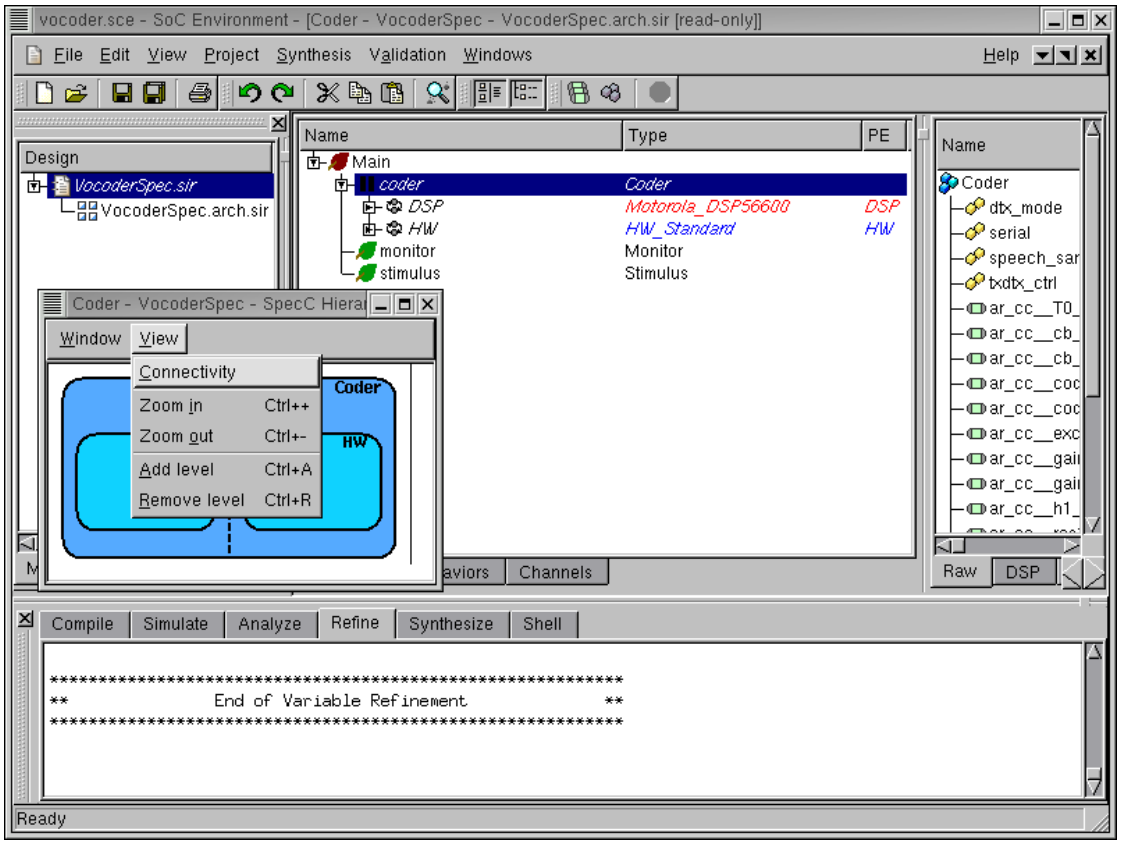
Since the top level behavior is "Coder", the test bench behaviors are not changed during architecture refinement. Therefore let’s select "Coder" by clicking in the corresponding row in the design window. We would like to see how the design looks when it is mapped to the selected architecture. To view the hierarchy of the new "Coder" behavior, go to View⟶Chart.

3.2.6.1. Browse architecture model (cont'd)



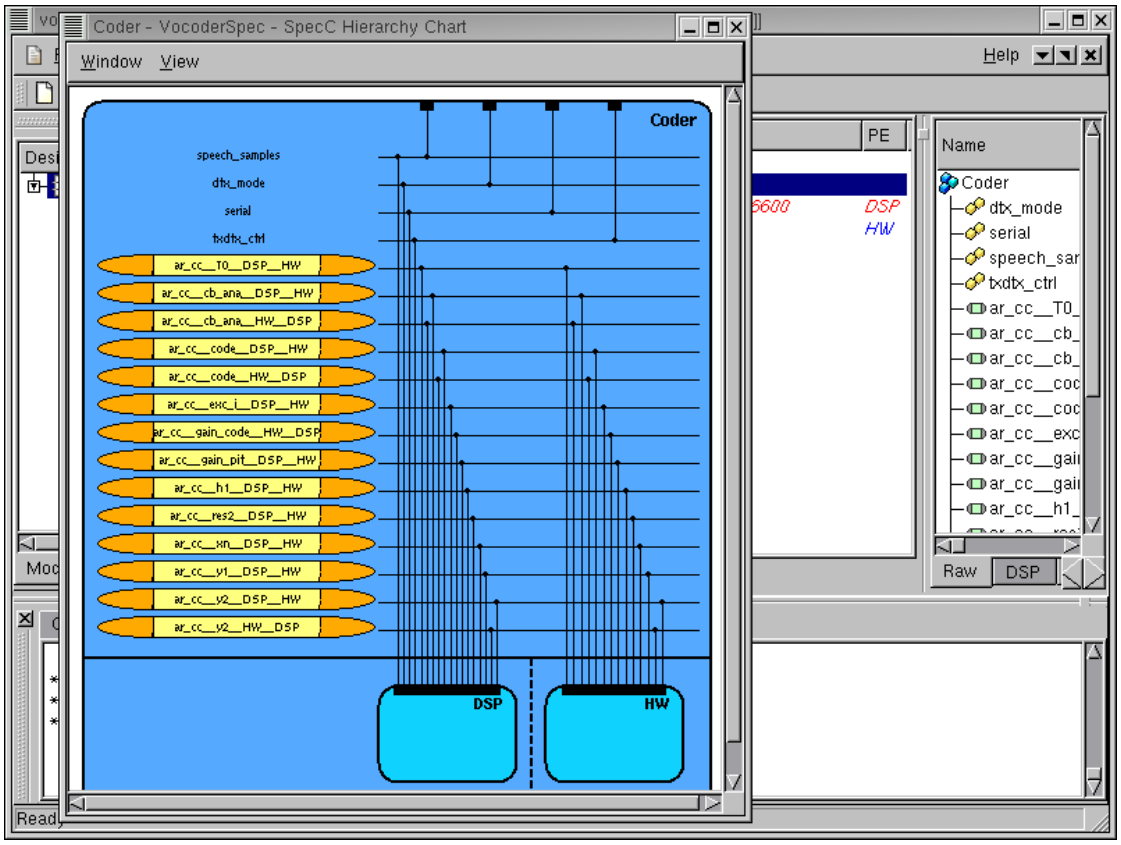
A window pops up, showing all sub-behaviors of the "Coder" behavior. As we can see, this new top level behavior Coder in the architecture model is composed of two new behaviors, "DSP" and "HW", which were constructed and inserted during architecture refinement. These behaviors at the top level indicate the presence of two components selected in the architecture. Note that they are also composed in parallel, which represents the actual semantics of the architecture model.

3.2.6.2. Browse architecture model (cont'd)



We would now like to see how the "DSP" and "HW" behaviors are communicating. This will verify if the refinement process was correctly executed. Go to View⟶Connectivity to see the connectivity between the "DSP" and the "HW" components.

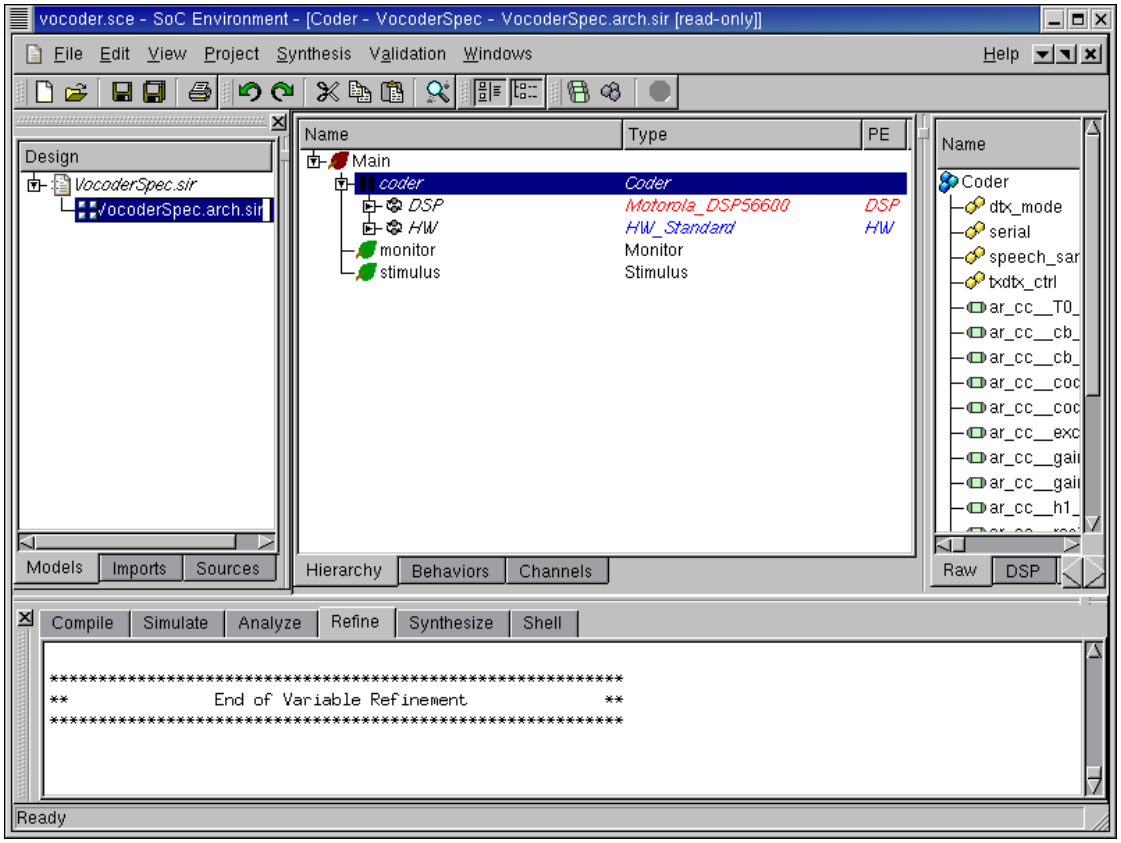
3.2.6.3. Browse architecture model (cont'd)



Enlarge the new window and scroll down to view the connectivity of the two components. We can see that "DSP" and "HW" components are connected through global variable channels, which were inserted during the architecture refinement. This is different from the original specification model, where only global variables were used for communication.

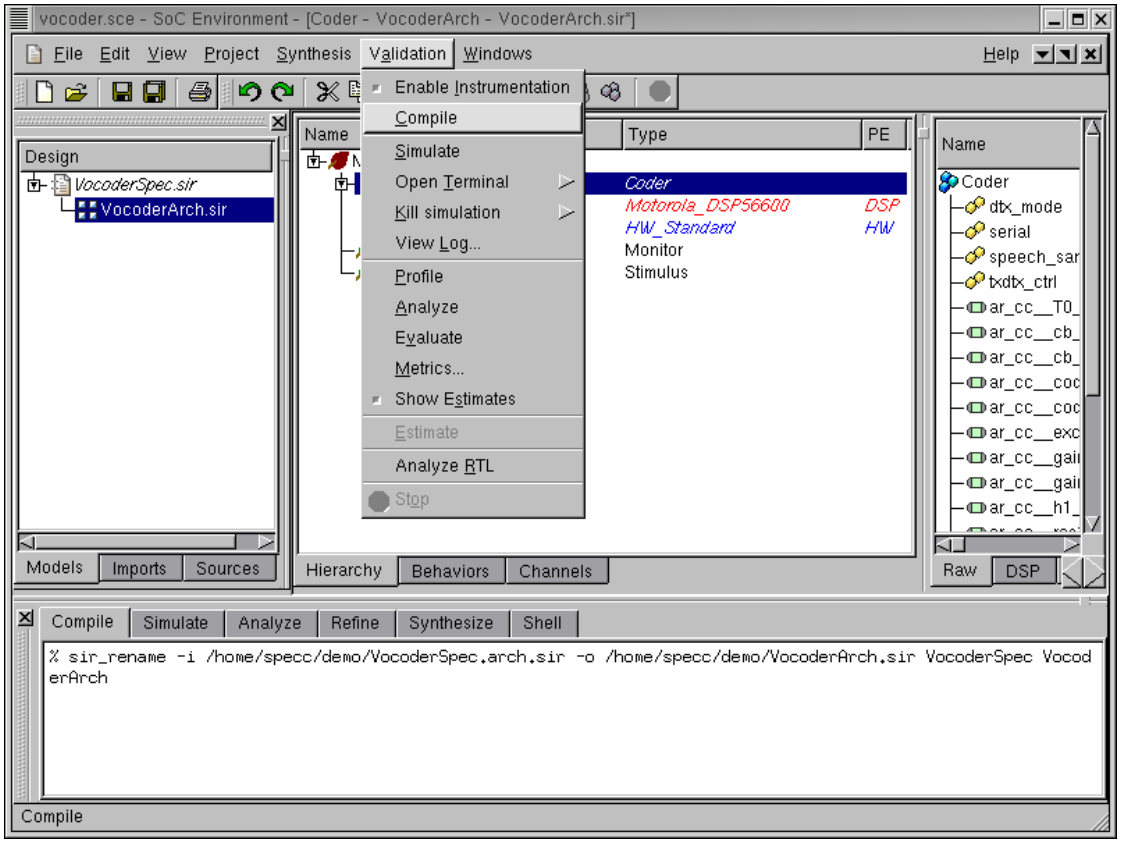
After checking the new architecture model, we can close the pop up window and go back to the design window by selecting Window⟶Close from the menu bar.

3.2.6.4. Rename architecture model



Like what we did for the specification model, we also change the name of the new model to be "VocoderArch.sir" in the project window. The renaming is just for the purpose of maintaining a nomenclature schema and to correctly identify the individual models.

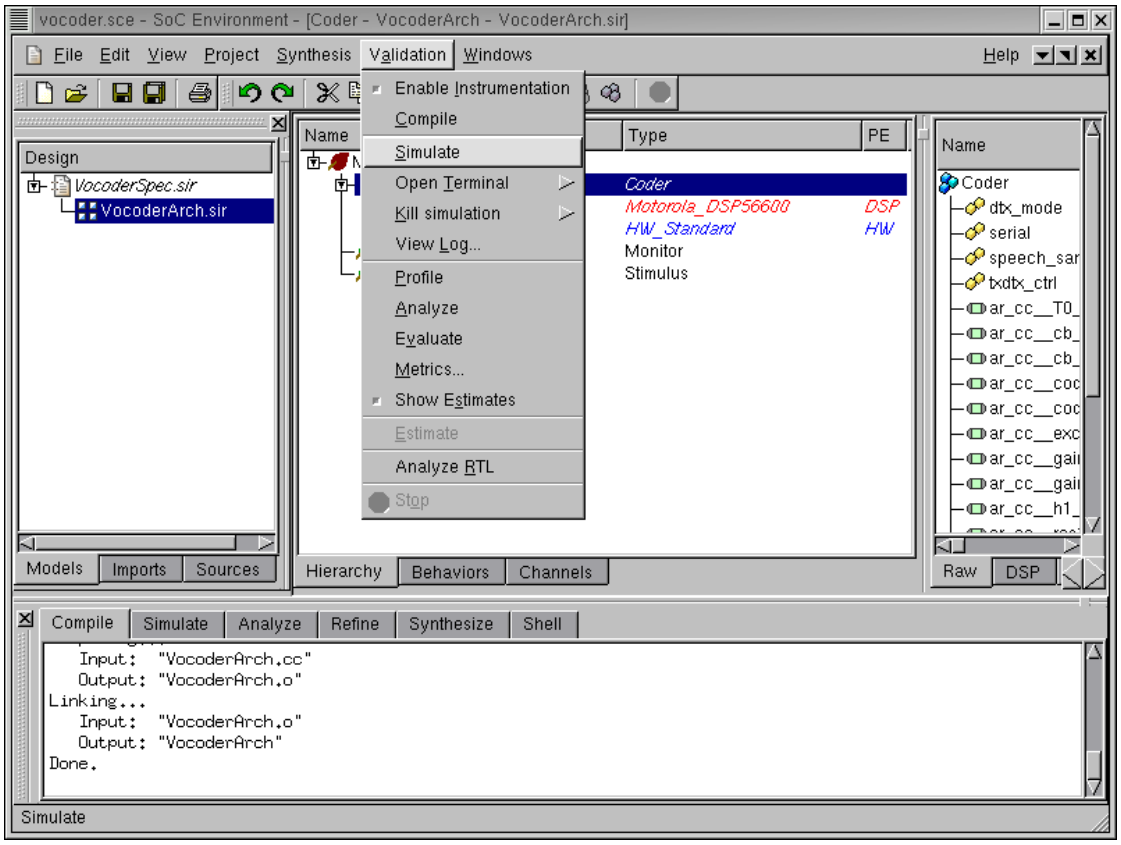
3.2.7. Simulate architecture model (optional)



This section shows the simulation of the generated architecture model. If the reader is not interested, she or he can skip this section and go directly to Section 3.3 *Software Scheduling and RTOS Model Insertion* (page 96).

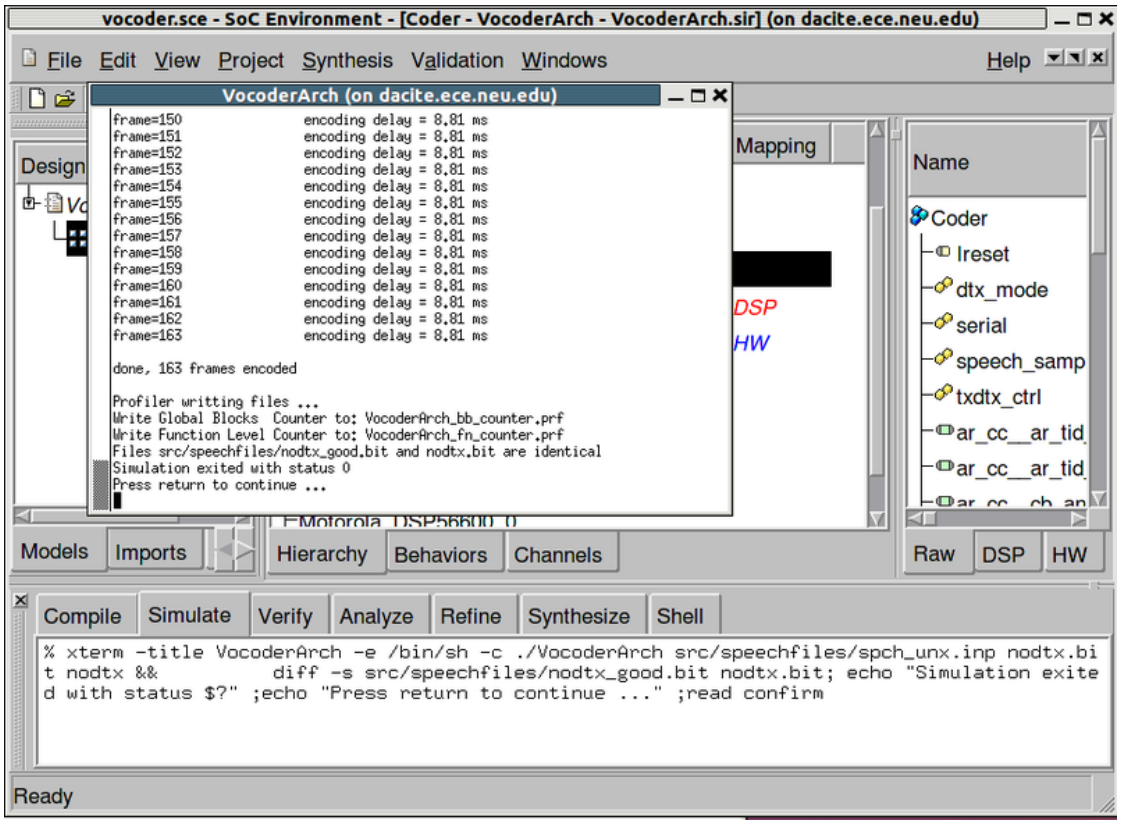
So far we have graphically visualized the automatically generated architecture. We have seen that in terms of its structural composition, the model meets the semantics of an architecture level model in our SoC methodology. However, we also need to confirm that the model has not lost any of its functionality in the refinement process. In other words the new model must be functionally equivalent to the specification. We will validate the architecture model through simulation. But first we need to compile the model into an executable. To compile the architecture model to executable, select Validation⟶Compile from the menu bar.

3.2.7.1. Simulate architecture model (optional) (cont'd)



The messages in the logging window show that the architecture model is compiled successfully without any syntax error. Now in order to verify that it is functionally equivalent to the specification model, we will simulate the compiled architecture model on the same set of speech data used in the specification validation by selecting Validation⟶Simulate from the menu bar.

3.2.7.2. Simulate architecture model (optional) (cont'd)



The simulation run is displayed in a new terminal window. As we can see, the architecture model was simulated successfully for all 163 frames speech data. The result bit file is also compared with the expected golden output given with the Vocoder standard. We have thus verified that the generated architecture model is functionally correct. In addition, the simulation of the architecture model shows that the processing time for each frame is 8.81 ms, which was not available when simulating the specification model.

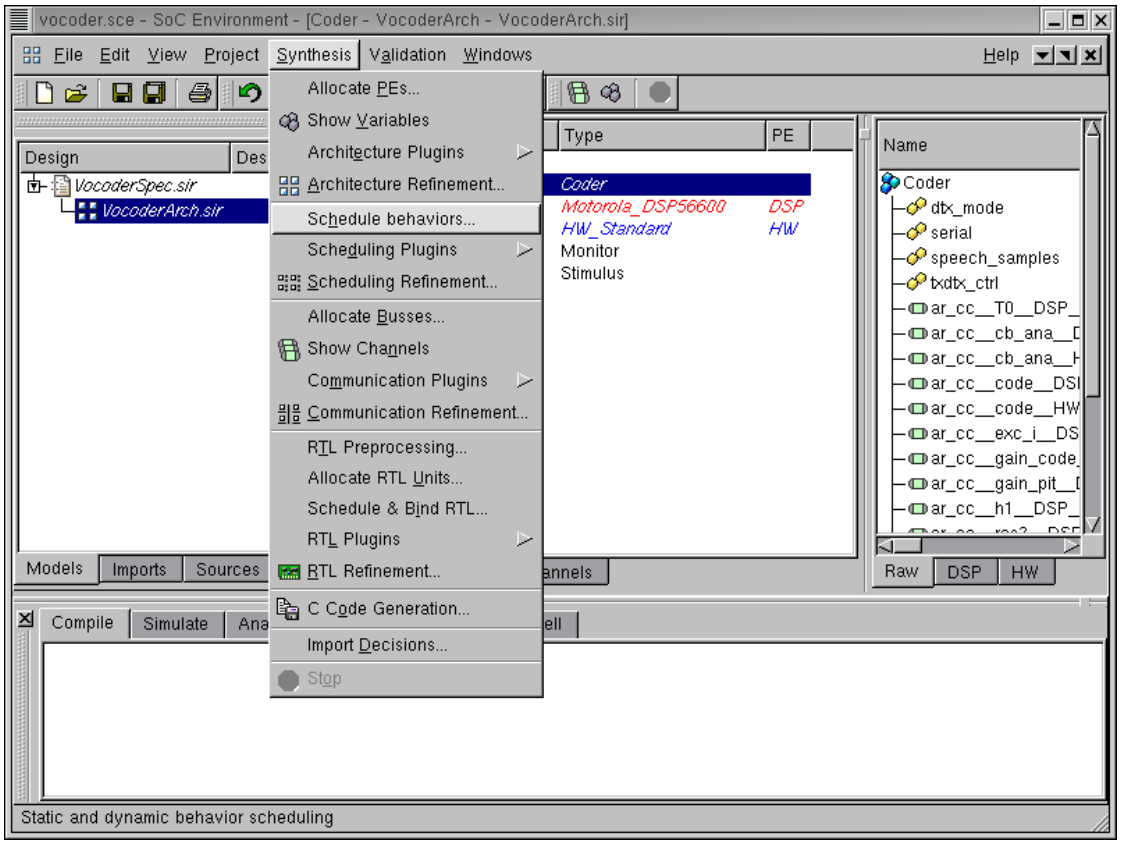
It must be noted as before that the testing process requires fairly intensive execution, but for the demo purposes we will omit multiple simulations and just show the concept. This concludes the step of architecture exploration.

3.3. Software Scheduling and RTOS Model Insertion

The next step in the system level design process is the serialization of behavior execution on the processing elements. Processing elements (PEs) have a single thread of control only. Therefore, behaviors mapped to the same PE can only execute sequentially and have to be scheduled. Software scheduling and RTOS model insertion is the design step to schedule the behaviors inside each PE.

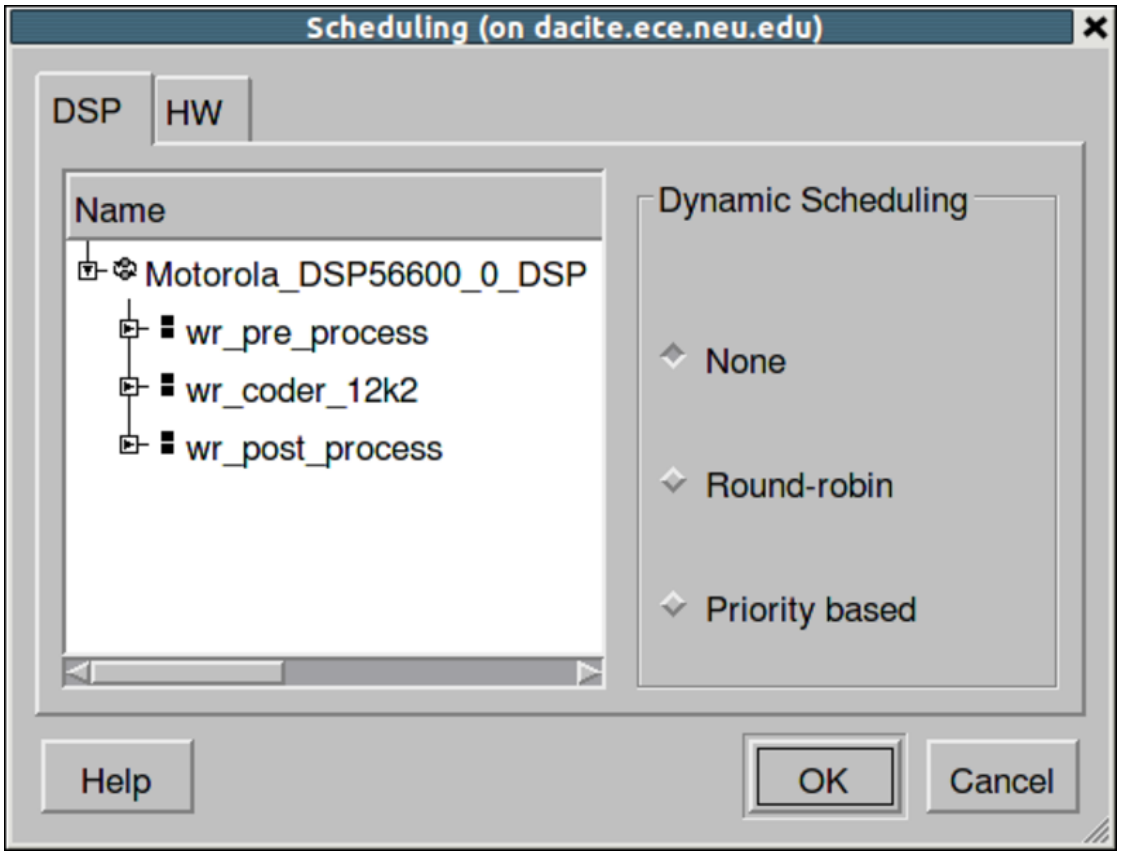
Depending on the nature of the PE and the data inter-dependencies, behaviors are scheduled statically or dynamically. In a static scheduling approach, behaviors are executed in a fixed and predetermined order, possibly flattening parts of the behavioral hierarchy. In a dynamic scheduling approach on the other hand, the order of execution is determined dynamically during runtime. Behaviors are arranged into potentially concurrent tasks. Inside each task, behaviors are executed sequentially. A RTOS model is inserted into the design. The RTOS model maintains a pool of task behaviors and dynamically selects a task to execute according to its scheduling algorithm. In this chapter we see how we make scheduling decisions using SCE.

3.3.1. Serialize behaviors



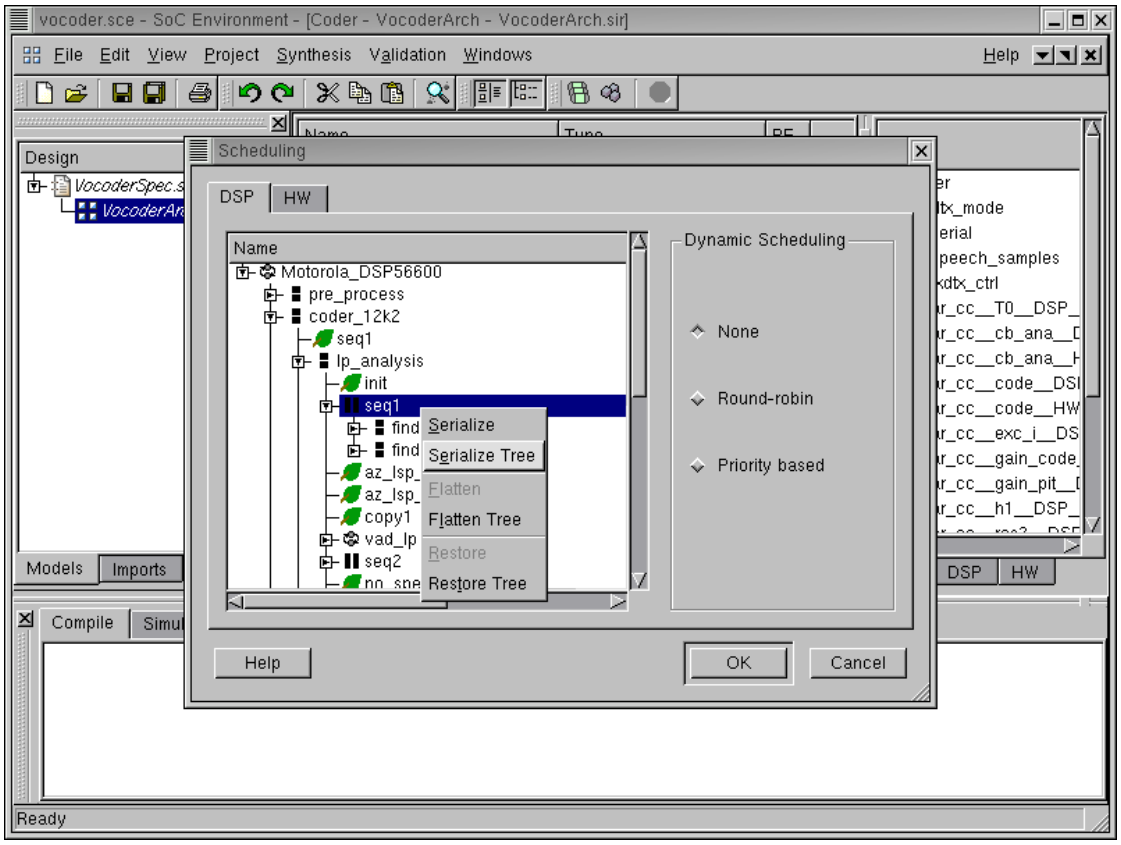
To start behavior scheduling, select Synthesis⟶Schedule behaviors from the menu bar.

3.3.1.1. Schedule software



A Scheduling window will pop up. This window includes scheduling options for two PEs (DSP and HW). We begin by selecting the scheduling algorithm for the software. We can do either static scheduling or dynamic scheduling for the software. In case of dynamic scheduling, a RTOS model corresponding to the selected scheduling strategy is imported from the library and instantiated in the PE. The RTOS model provides an abstraction of the key features that define a dynamic scheduling behavior independent of any specific RTOS implementation. SCE provides two RTOS models with different dynamic scheduling algorithms: round-robin and priority based.

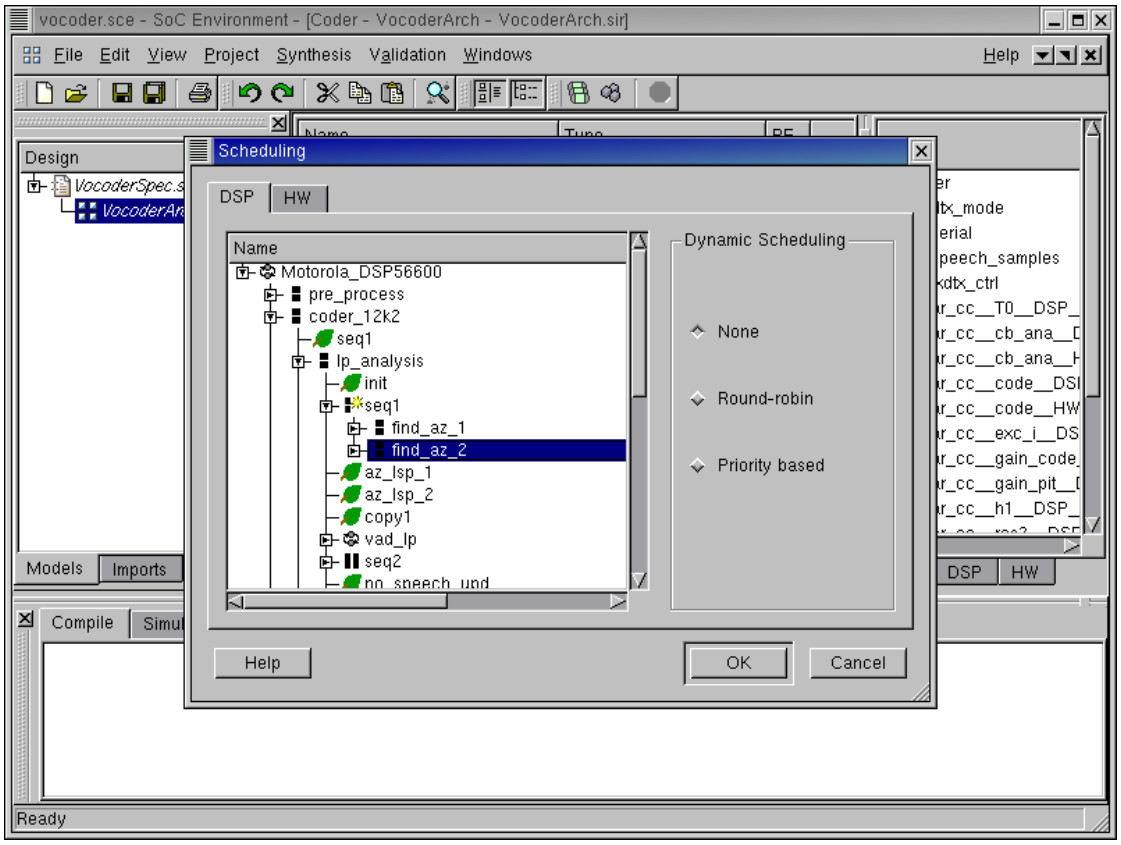
3.3.1.2. Schedule software (cont'd)



Behavior scheduling is done by converting all concurrent SpecC "par" or "pipe" statements into sequential statements. This conversion is achieved by performing the "serialize" operations on the intended behaviors. For example, assume that behavior "A" is a "par" composition of behavior "B" and "C". With a "serialize" operation, behavior "A" will be changed to a sequential execution of "B" and "C" by default. Another kind of operations, "flatten" are often performed during behavior scheduling to change the behavior hierarchy. Continuing with our example, if behavior "B" itself is composed of "D" and "E" in parallel, a "flatten" operation on "B" removes "B" from "A" while promoting its sub-behaviors, "D" and "E" one level up. As the result, behavior "A" becomes a "par" composition of "D", "E" and "C". Note that the hierarchy relation among behaviors is most conveniently represented as a tree, operations "serialize tree" and "flatten tree" are also provided by SCE to serialize or flatten behaviors of a subtree recursively.

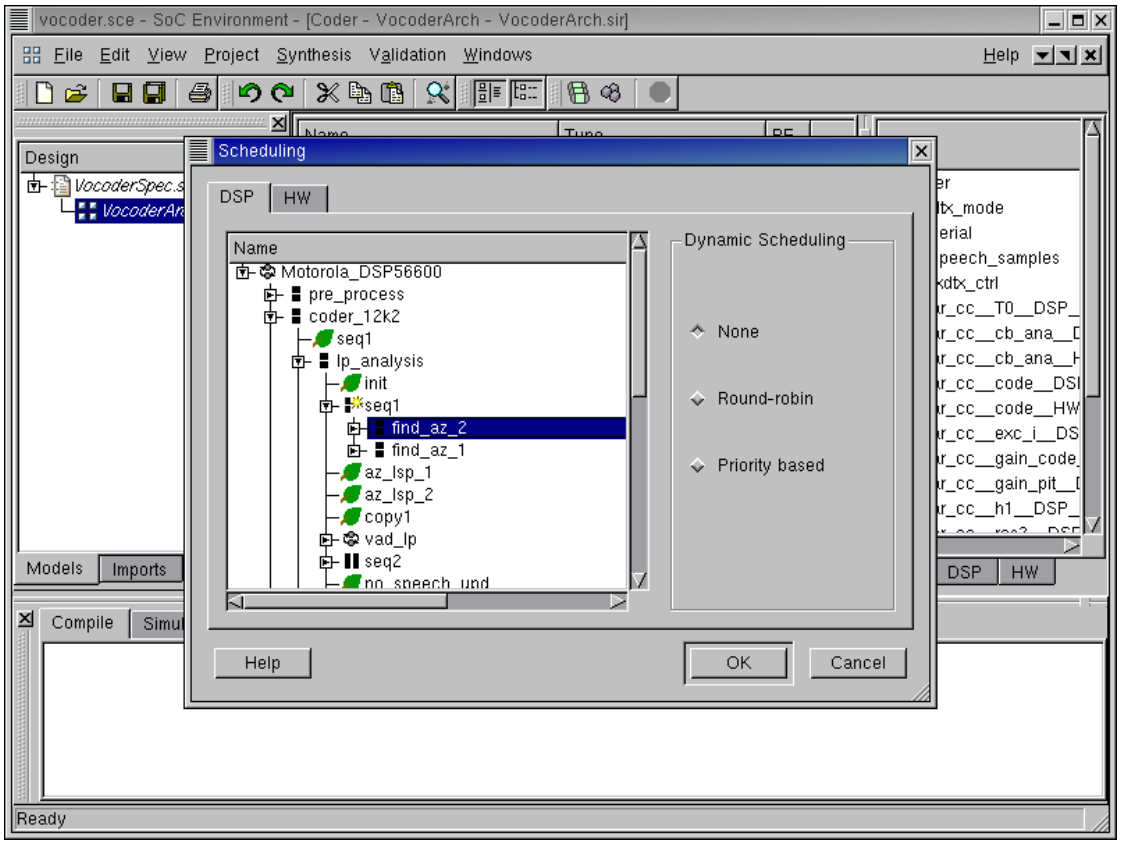
In our design, for example, to serialize the sub-behaviors of behavior "seq1", in the design hierarchy tree, select behavior "seq1". Right click to bring up a menu window and select Serialize Tree from the menu.

3.3.1.3. Schedule software (cont'd)



Now that the two parallel child behaviors of behavior "seq1": behavior "find_az_1" and behavior "find_az_2" are converted into two sequential behaviors. We can see that behavior "find_az_1" is executed before behavior "find_az_2". This execution order is created by the tool. The designer can modify the execution order.

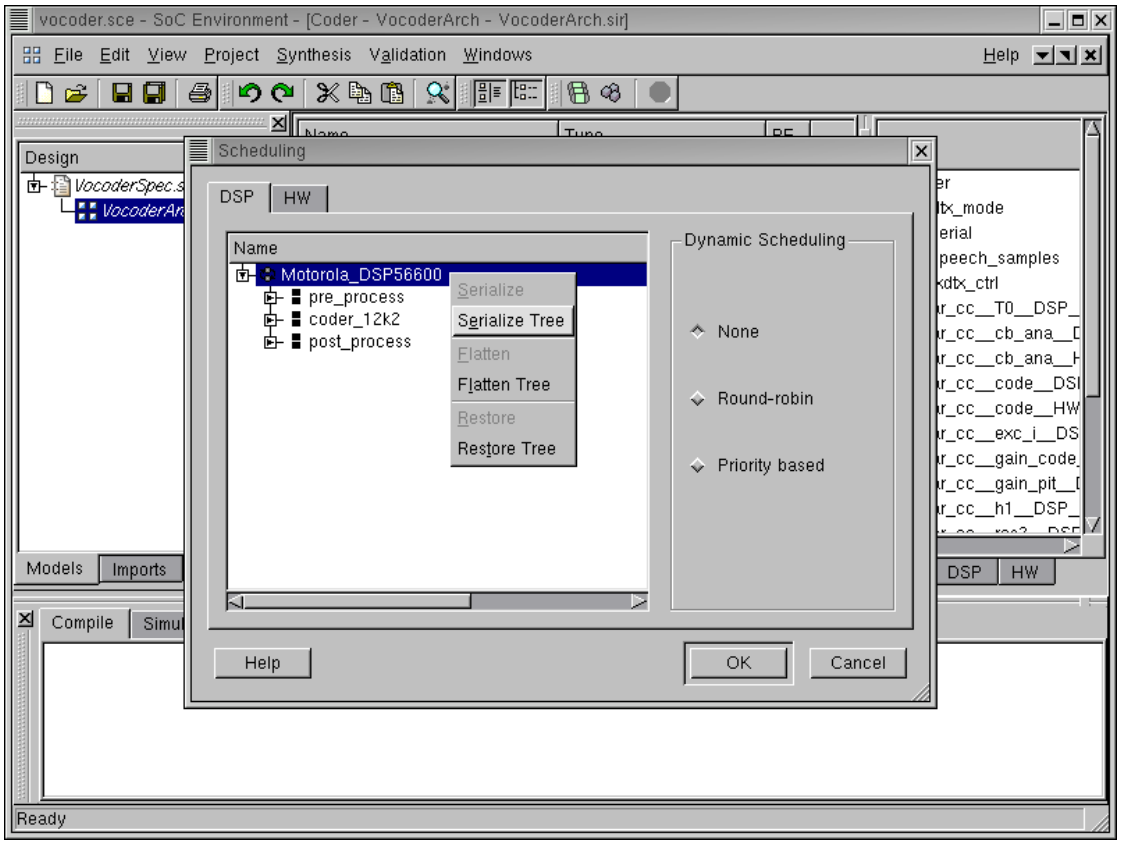
3.3.1.4. Schedule software (cont'd)



Select behavior "find_az_2". Left click and move behavior "find_az_2" before behavior "find_az_1". Now behavior "find_az_2" is executed before "find_az_1". In general, the designer can specify any "par" or "pipe" statements to be scheduled and manually specify the execution order of any parallel behaviors in the same level. The remaining parallel behaviors can either be dynamically scheduled by the RTOS model or statically serialized by the tool.

Since we want the tool to schedule all the behaviors automatically, we restore the execution order created by the tool. Select behavior "find_az_1". Left click and move behavior "find_az_1" before behavior "find_az_2".

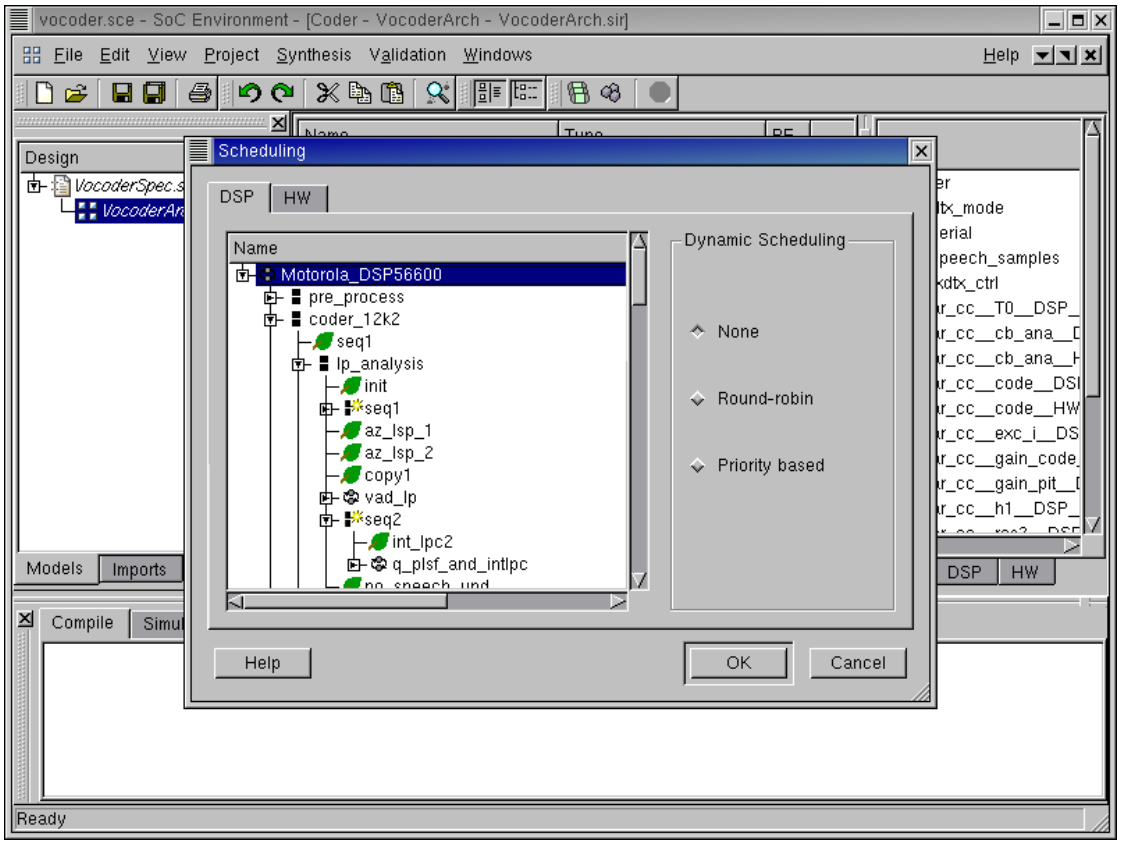
3.3.1.5. Schedule software (cont'd)



For our example, since there are not many parallel behaviors in DSP, we statically schedule the behaviors in DSP. In the dynamic scheduling box, click and select None.

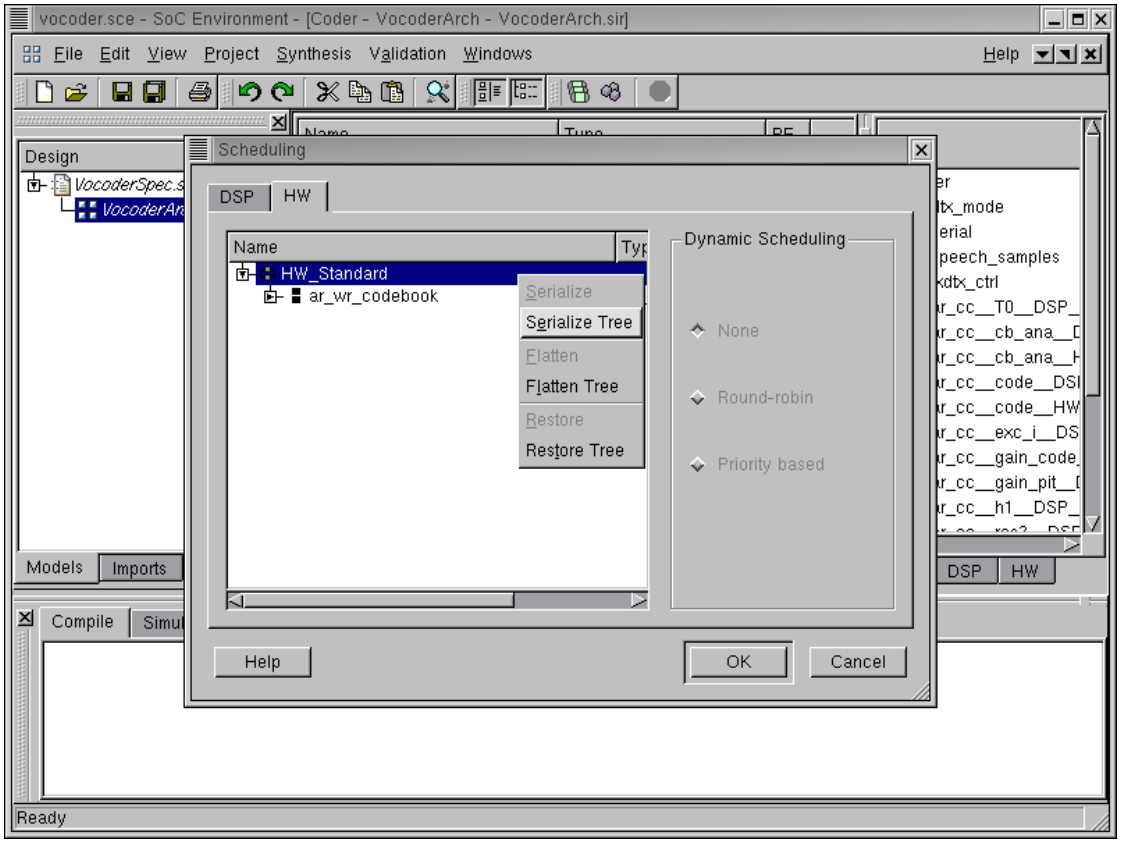
Also, we will leave the decision of behavior execution order to be made automatically by the tool. In the design hierarchy tree, select behavior "Motorola_DSP56600". Right click and select Serialize Tree.

3.3.1.6. Schedule software (cont'd)



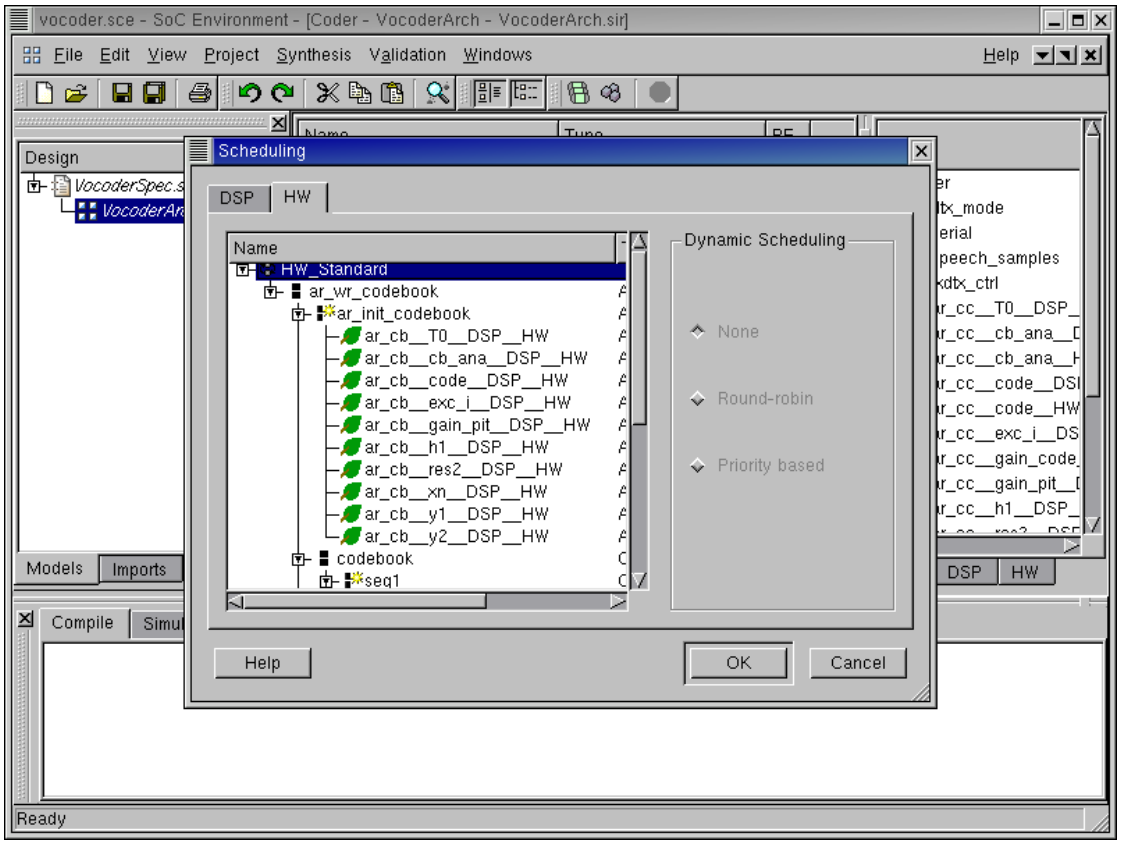
As shown in the figure, all the child behaviors of behavior "Motorola_DSP56600" are serialized. Behaviors that are modified as a result of serialization are marked with a "*" symbol next to them.

3.3.1.7. Serialize behaviors in HW



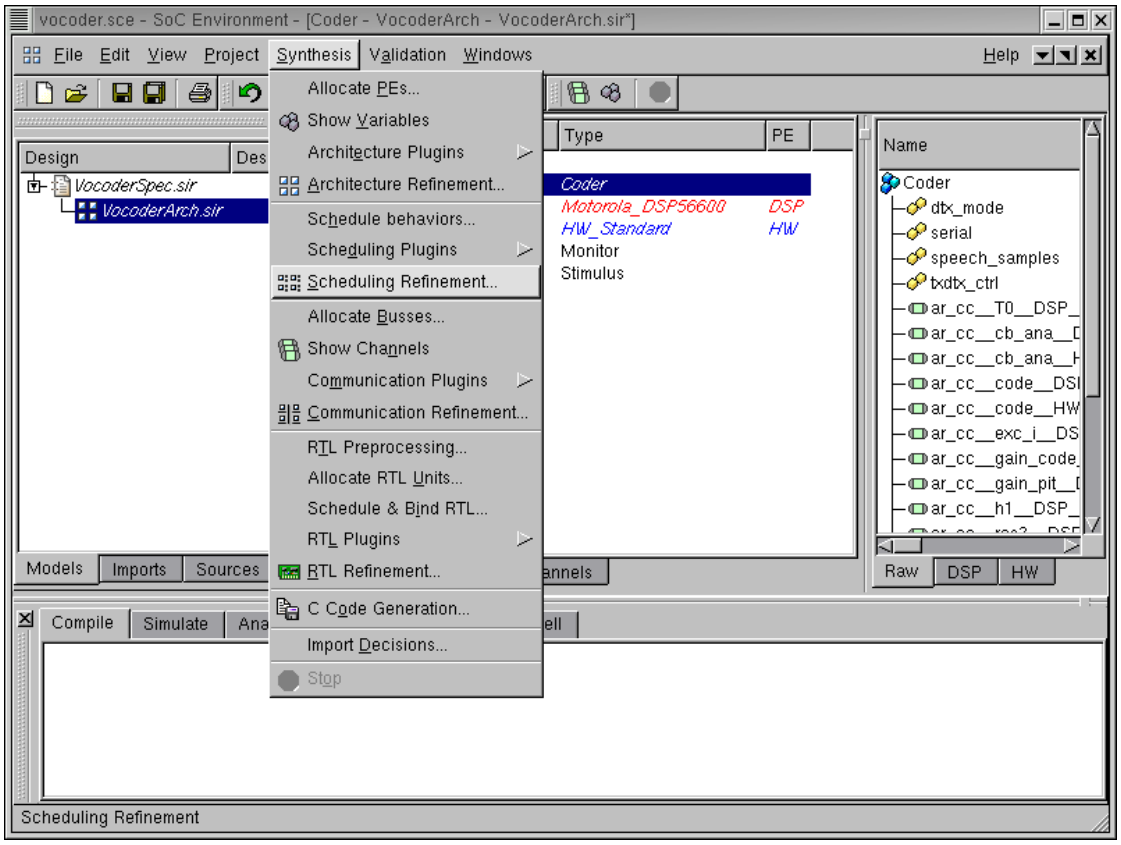
The next step is to serialize behaviors in HW. Since custom hardware can only be statically scheduled, the dynamic scheduling box is disabled for HW. Click and select HW in the Scheduling window. In the design hierarchy tree, select behavior "HW_Standard". Right click and select Serialize Tree.

3.3.1.8. Serialize behaviors in HW (cont'd)



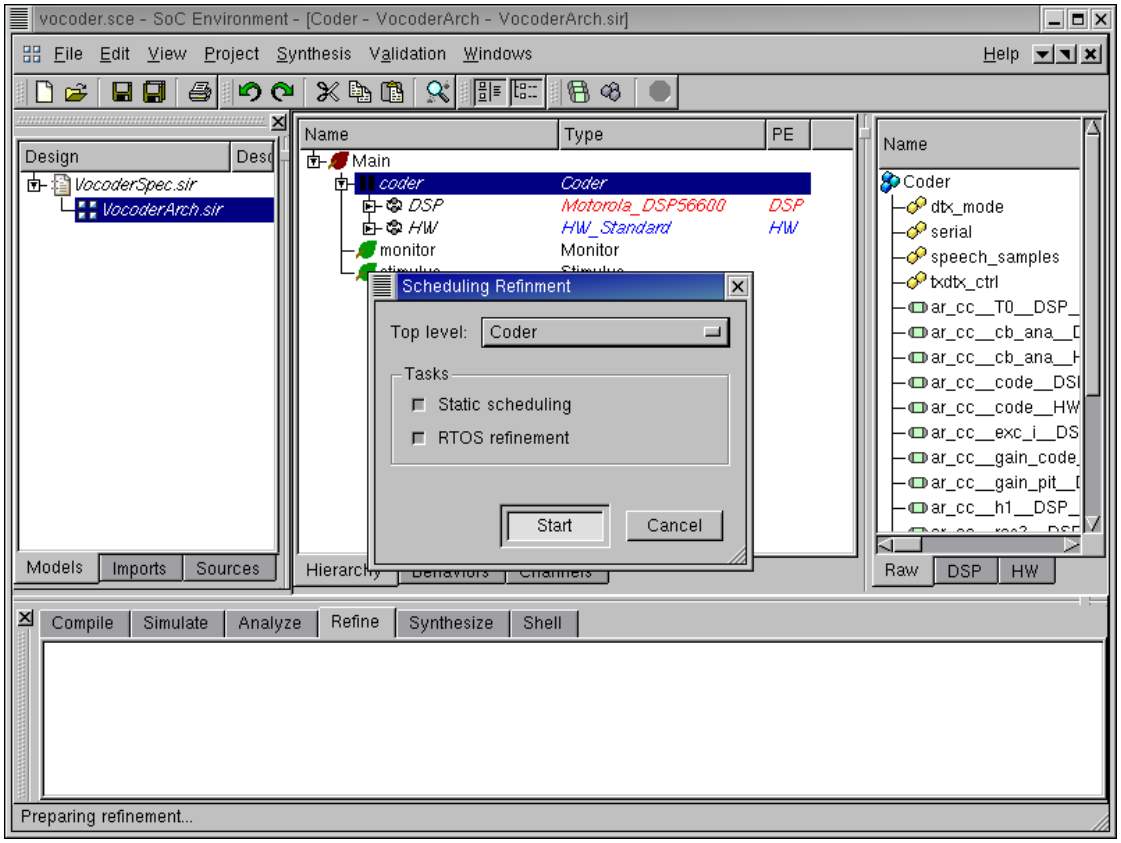
As shown in the figure, all the child behaviors of behavior "HW_Standard" are serialized. Click OK button to confirm the scheduling decision.

3.3.2. Generate serialized model



Once the scheduling decisions have been made, we can refine the architecture model to reflect the changes. A software scheduling and RTOS model insertion tool is integrated in SCE. The tool will generate the model to reflect the scheduling algorithm we selected. In case of dynamic scheduling, a RTOS model is inserted into the design and behaviors are converted into tasks with assigned priorities. To invoke the tool, go to Synthesis menu and select Scheduling Refinement.

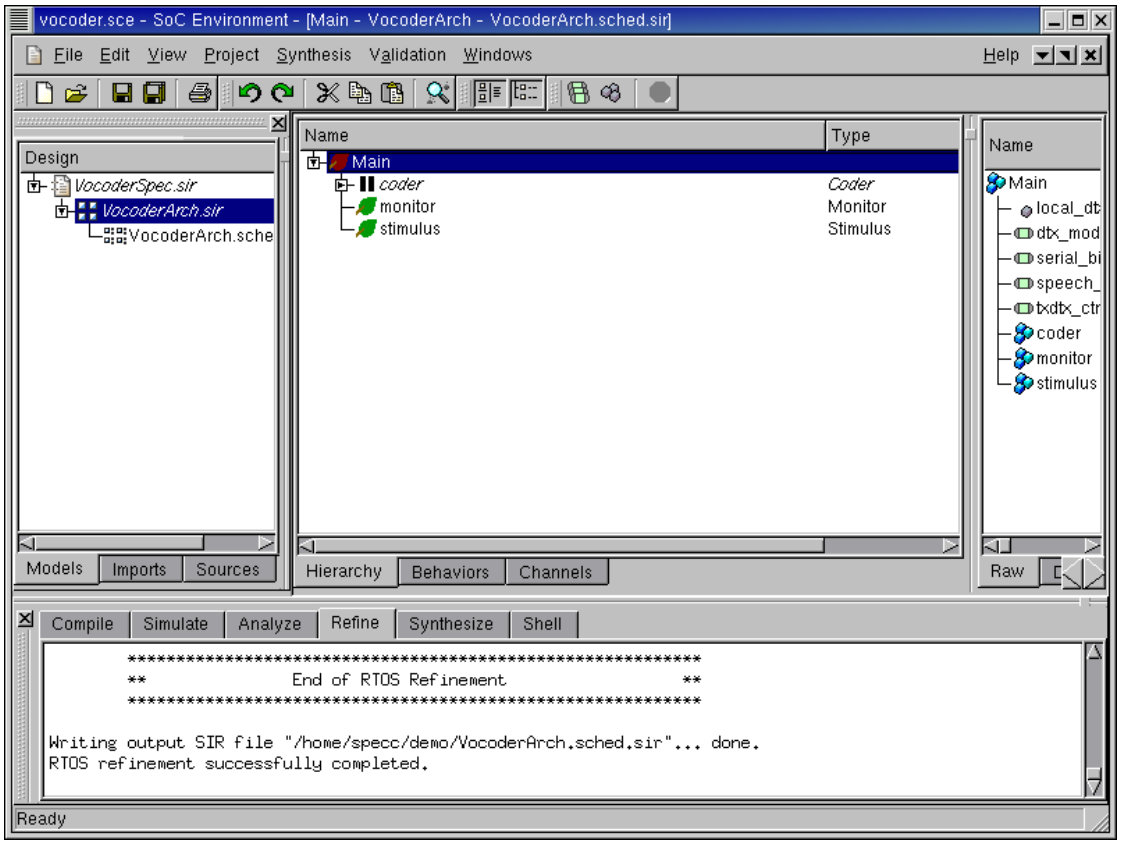
3.3.2.1. Refine after serialization



A dialog box pops up for selecting specific refinement tasks. By default, all tasks will be performed in one go. Press the Start button to start the refinement.

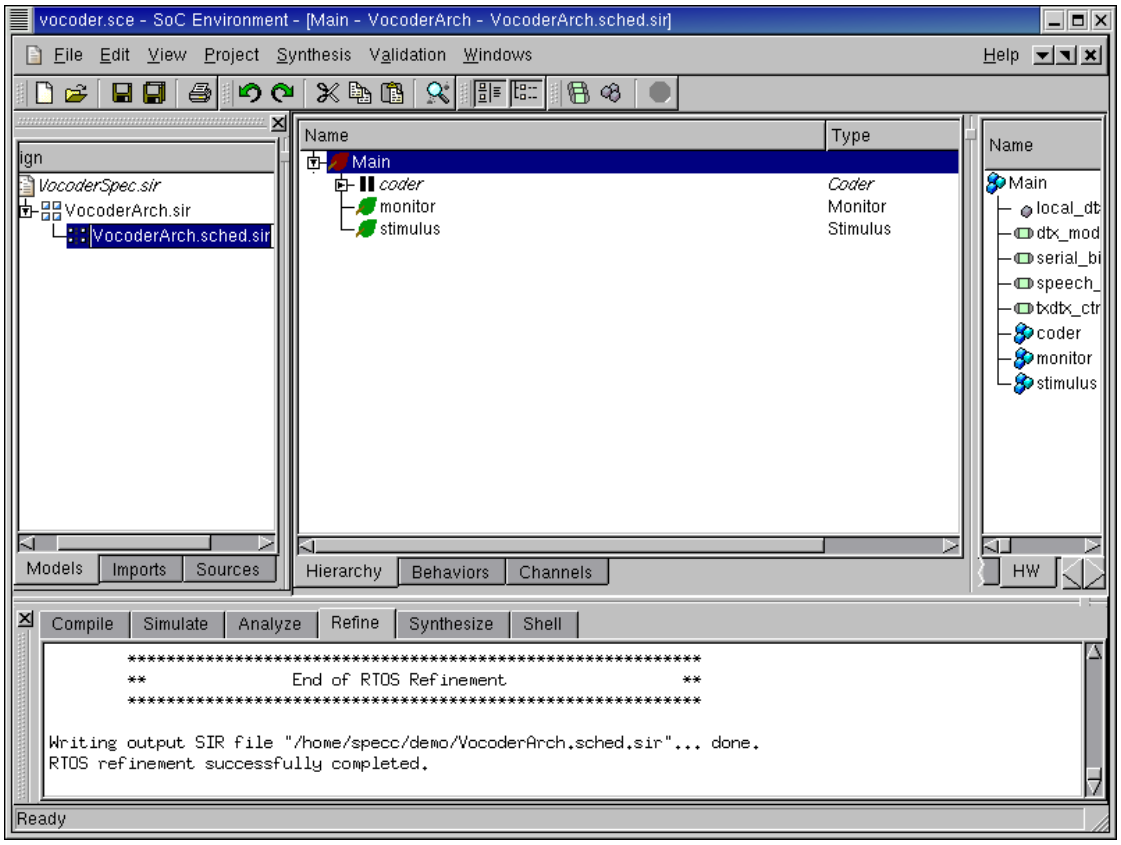
It must be noted that the user has an option to do the refinement tasks one step at a time. For instance, a designer may select only static scheduling if he or she is not concerned about observing the dynamic scheduling behavior on the component.

3.3.2.2. Refine after serialization (cont'd)



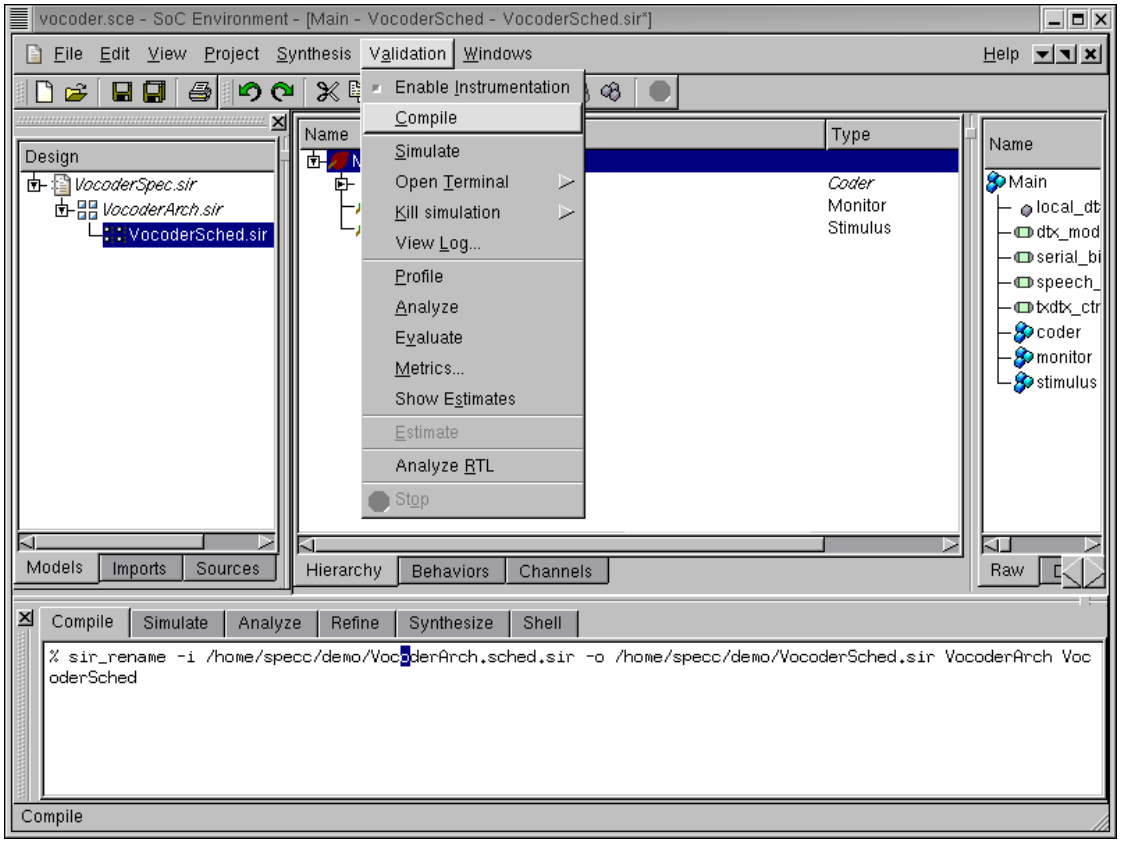
The logging window shows the refinement process. After the refinement, the newly generated serialized model "VocoderArch.sched.sir" is displayed to the design window. It is also added to the current project window, under the architecture model "VocoderArch.sir" to indicate that it was derived from "VocoderArch.sir".

3.3.2.3. Refine after serialization (cont'd)



As we did for previous models, we change the name of the serialized architecture model to "VocoderSched.sir" in the project window.

3.3.3. Simulate serialized model (optional)

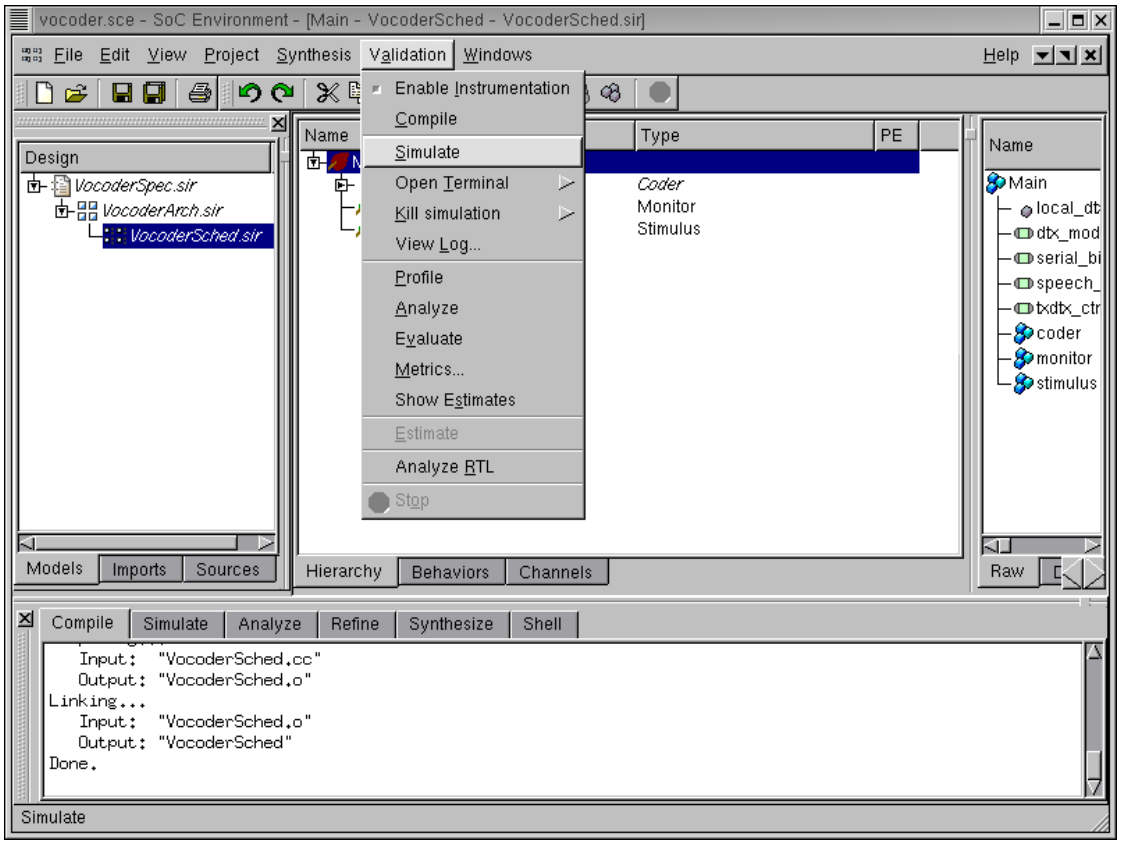


This section shows the simulation of the generated model. If the reader is not interested, she or he can skip this section and go directly to Section 3.4 *Communication Synthesis* (page 113).

Serialization refinement is now complete with the generation of a new model. However, we also need to confirm that the model has not lost any of its functionality in the refinement process. In other words the new model must be functionally equivalent to the architecture model.

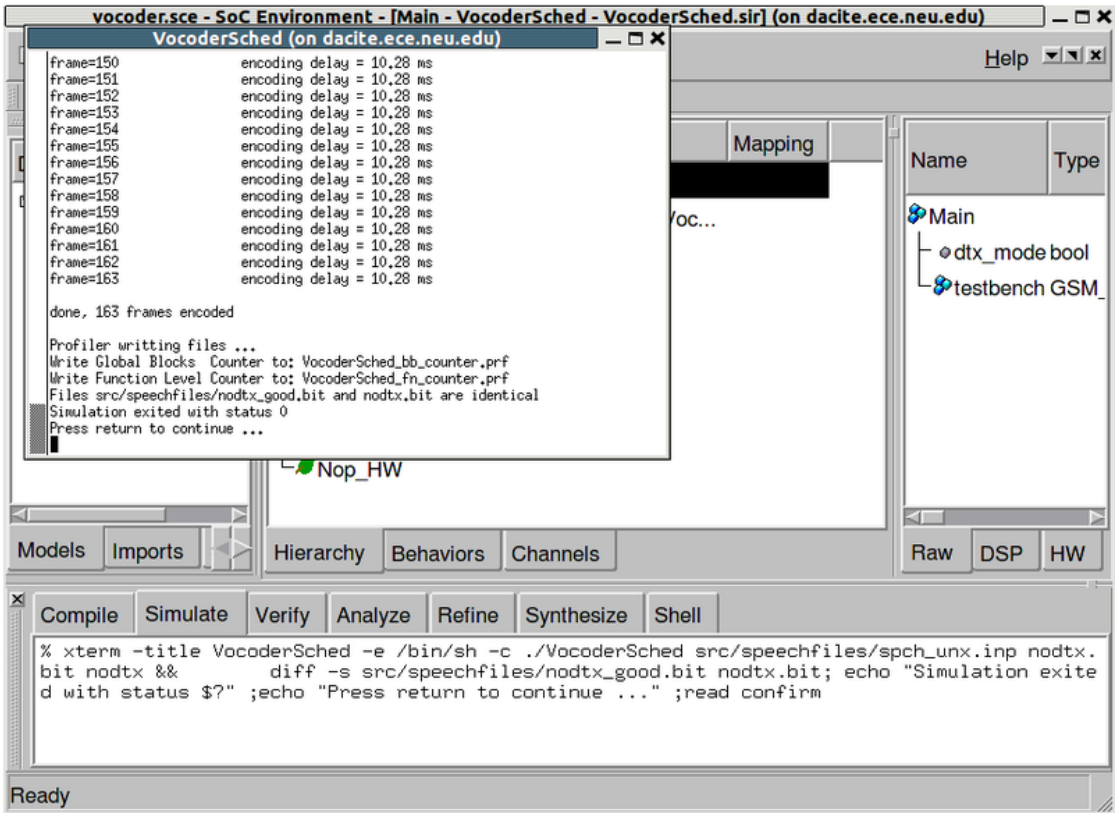
We will validate the serialized architecture model through simulation. But first we need to compile the model into an executable. To compile the serialized architecture model to executable, go to Validation menu and select Compile.

3.3.3.1. Simulate serialized model (optional) (cont'd)



The messages in the logging window shows that the refined model is compiled successfully without any errors. Now in order to verify that it is functionally equivalent to the architecture model, we will simulate the compiled model on the same set of speech data used in the specification validation. Go to Validation menu and select Simulate.

3.3.3.2. Simulate serialized model (optional) (cont'd)



The simulation run is displayed in a new terminal window. As we can see, the serialized architecture model was simulated successfully for all 163 frames of speech data. The result bit file is also compared with the expected golden output given with the Vocoder standard. We have thus verified that the generated refined model is functionally correct. Note that the execution time for each frame now becomes 10.28 ms. Recall that the execution time was 8.81 ms for each frame before the software scheduling is performed. The increase of execution time is reasonable since the concurrency in the previous model is removed by the software scheduling.

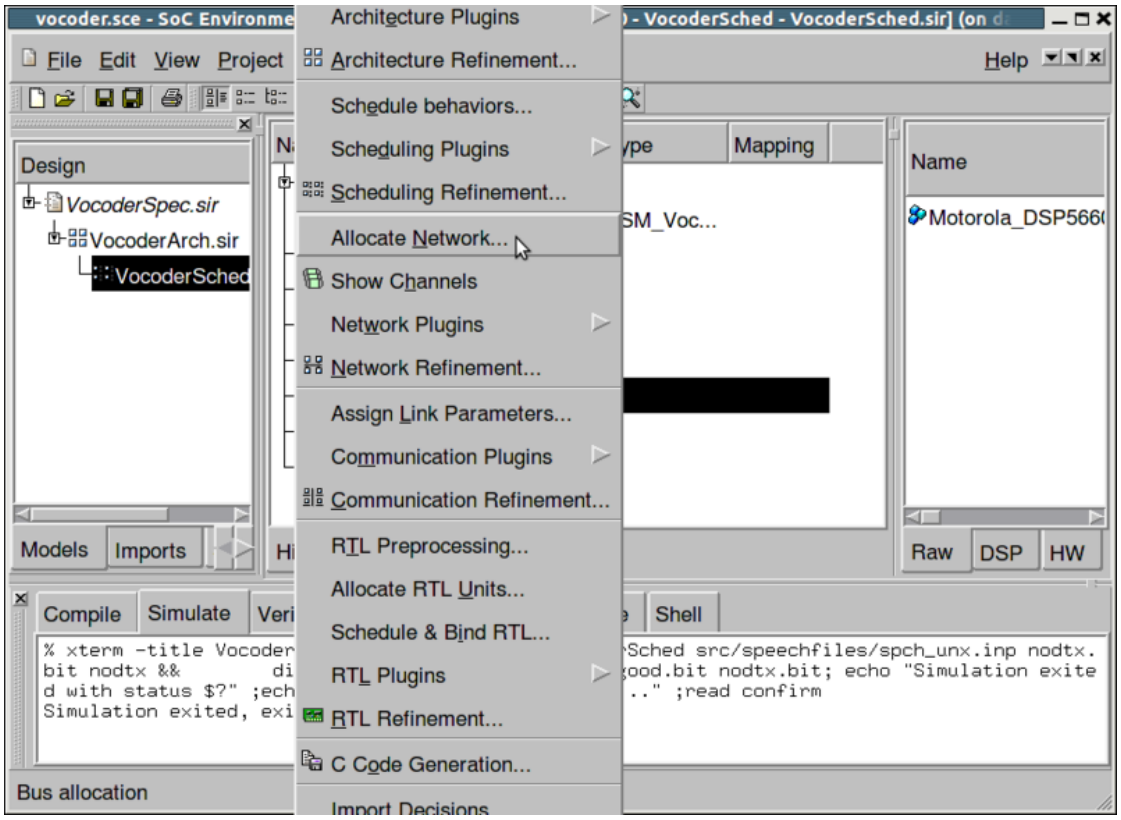
3.4. Communication Synthesis

Communication synthesis is the second part of the system level synthesis process. It refines the abstract communication between components in the architecture model. Specifically, the communication with variable channels is refined into an actual implementation over wires of the system bus. The steps involved in this process are as follows.

We begin with allocation of system buses and selection of bus protocols. A set of system buses is selected out of the bus library and the connectivity of the components with system buses is defined. In other words, we determine a bus architecture for our design.

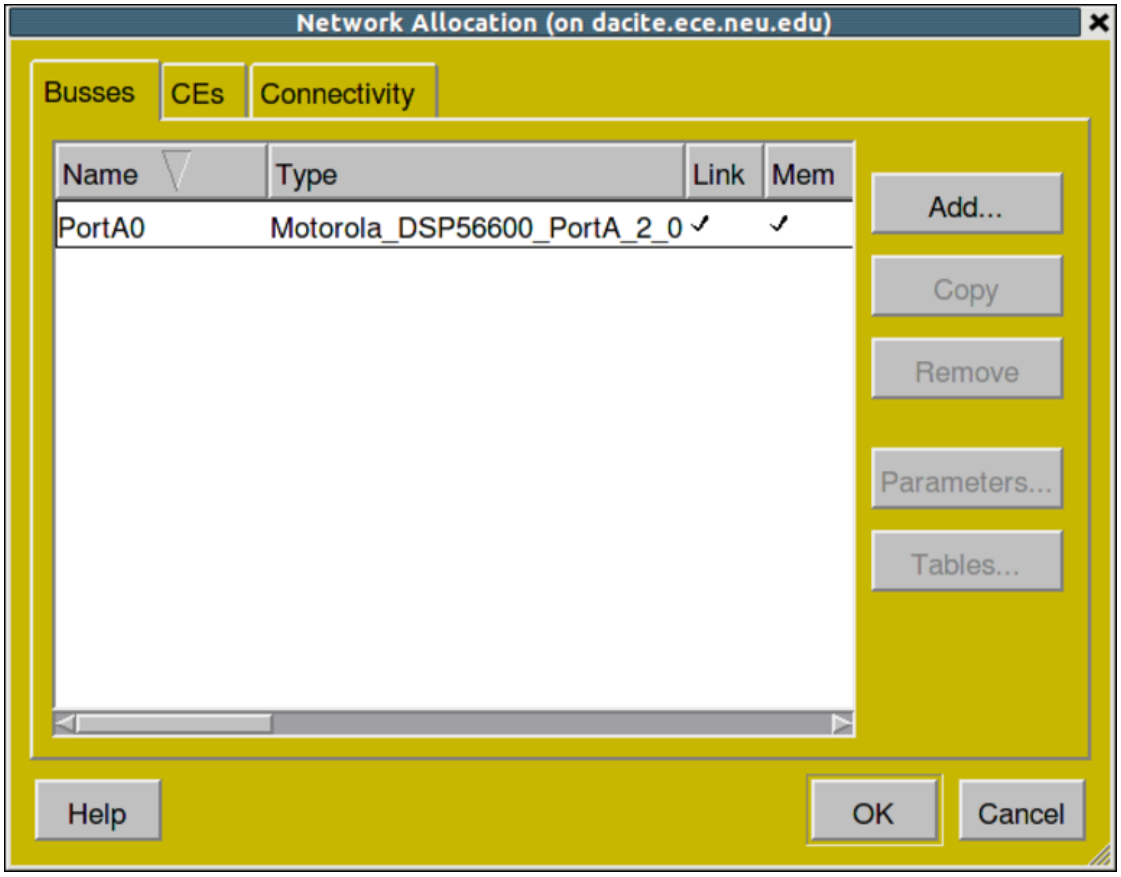
This is followed by grouping of abstract variable channels. The communication between system components has to be implemented with busses instead of variable channels. Thus these channels are grouped and assigned to the chosen system busses. Once this is done, the automatic refinement tool produces the required bus drivers for each component. It also divides variables into slices whose size is the same as width of the data bus. Therefore that each slice can be sent or received using the bus protocol. The entire variable is sent or received using multiple transfers of these slices.

3.4.1. Select bus protocols



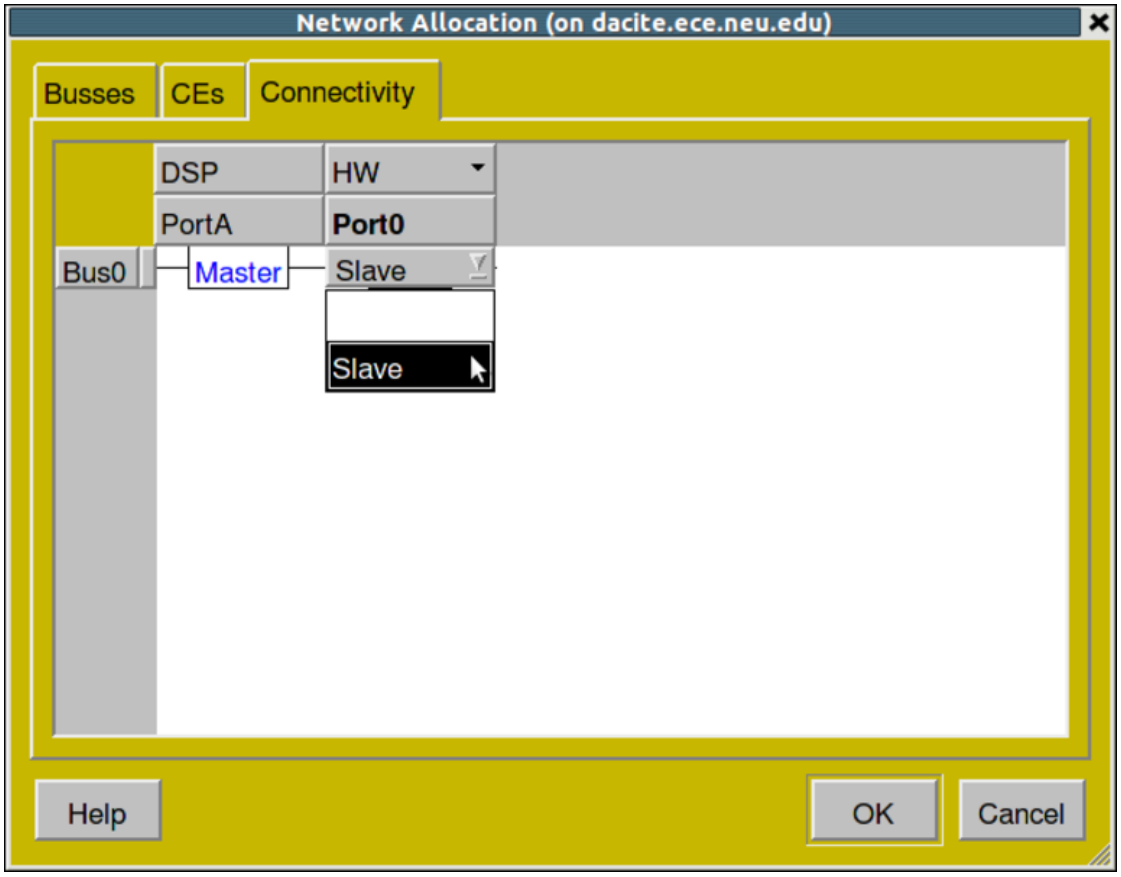
As explained earlier, we begin by selecting a suitable bus for our system. Note that in the presence of only two components, one bus would suffice. However, in general the user may select multiple buses if the need arises. Bus allocation is done by selecting Synthesis⟶Allocate Network from the menu bar.

3.4.1.1. Select bus protocols (cont'd)



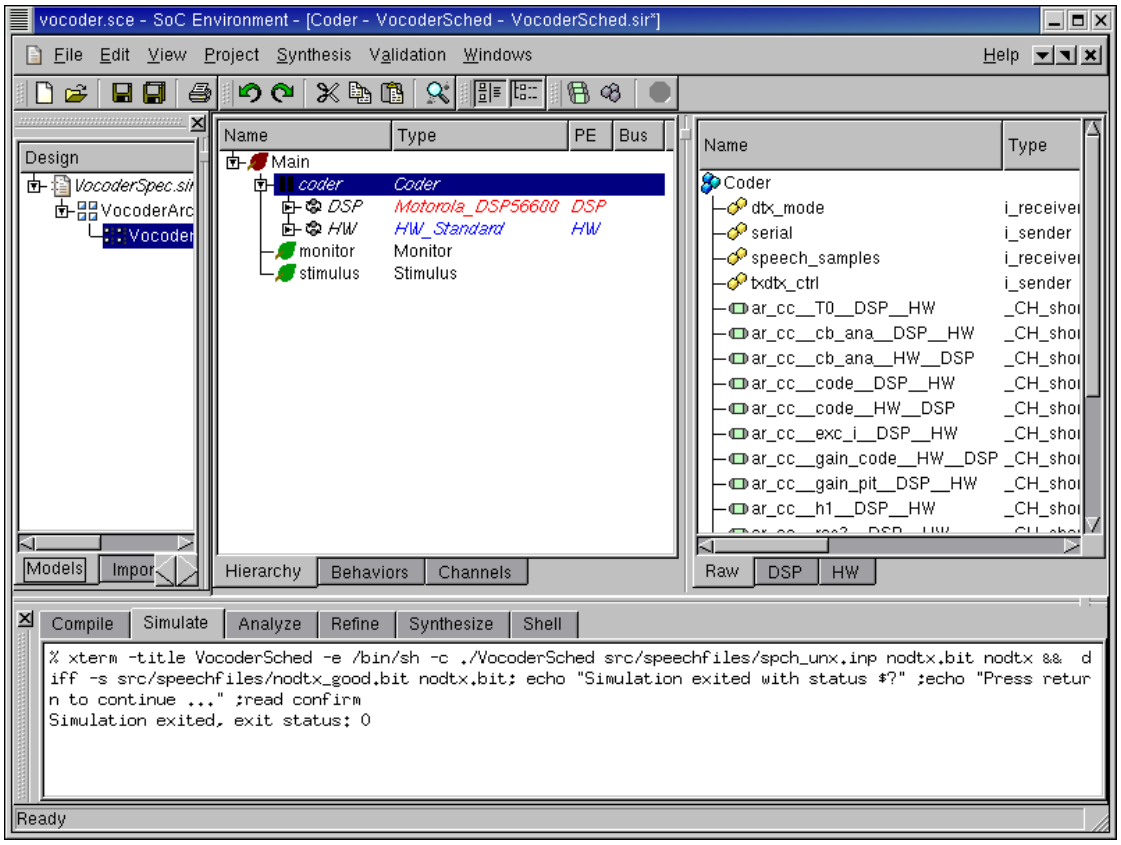
A Bus Allocation window pops up showing the bus allocation table. In the network allocation window there will be three tabs: Busses, CEs, and Connectivity. In the Busses tab, by default the "Motorola_DSP56600_PortA" bus will be present. This is the required bus. Rename this bus to "Bus0" by left clicking "Port0" and typing "Bus0." Note that the architecture chosen for the design has an impact on the selection of busses. More often that not, the primary component in the design dictates the bus selection process. In this case, we have a DSP with an associated bus. It makes sense for the designer to select that bus to avoid going through the overhead of generating a custom bus adapter. Click the Connectivity tab to allocate the bus.

3.4.1.2. Select bus protocols (cont'd)



The Connectivity tab shows that the hardware element has one port, PortA0, available. Set PortA0 of "HW" as "Slave on "Bus0" by clicking the menu under "HW" "Port0" and selecting slave. Click OK to confirm your changes.

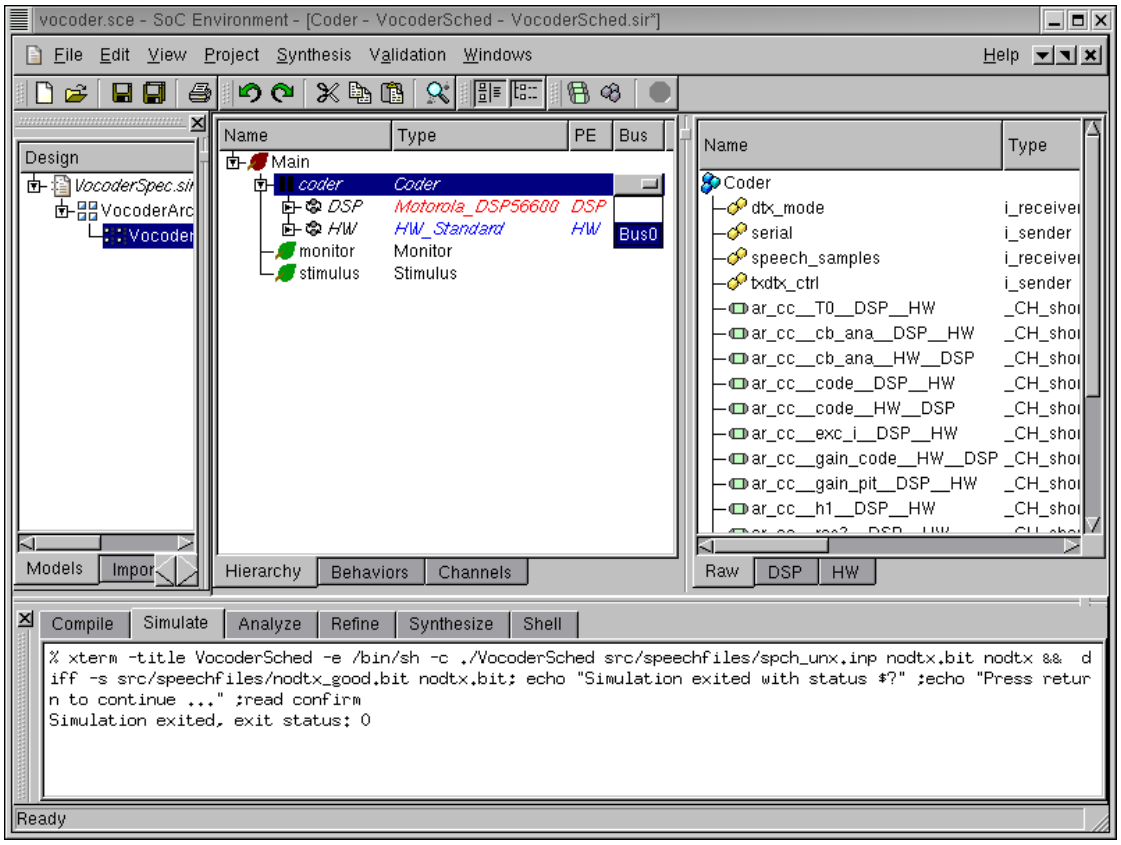
3.4.2. Map channels to buses



Once the bus allocation has been done, we need to group the channels of the architecture model and assign them to the system buses. Recall that in the architecture model, we had communication between components with abstract variable channels. We now have to assign those variable channels to the system bus.

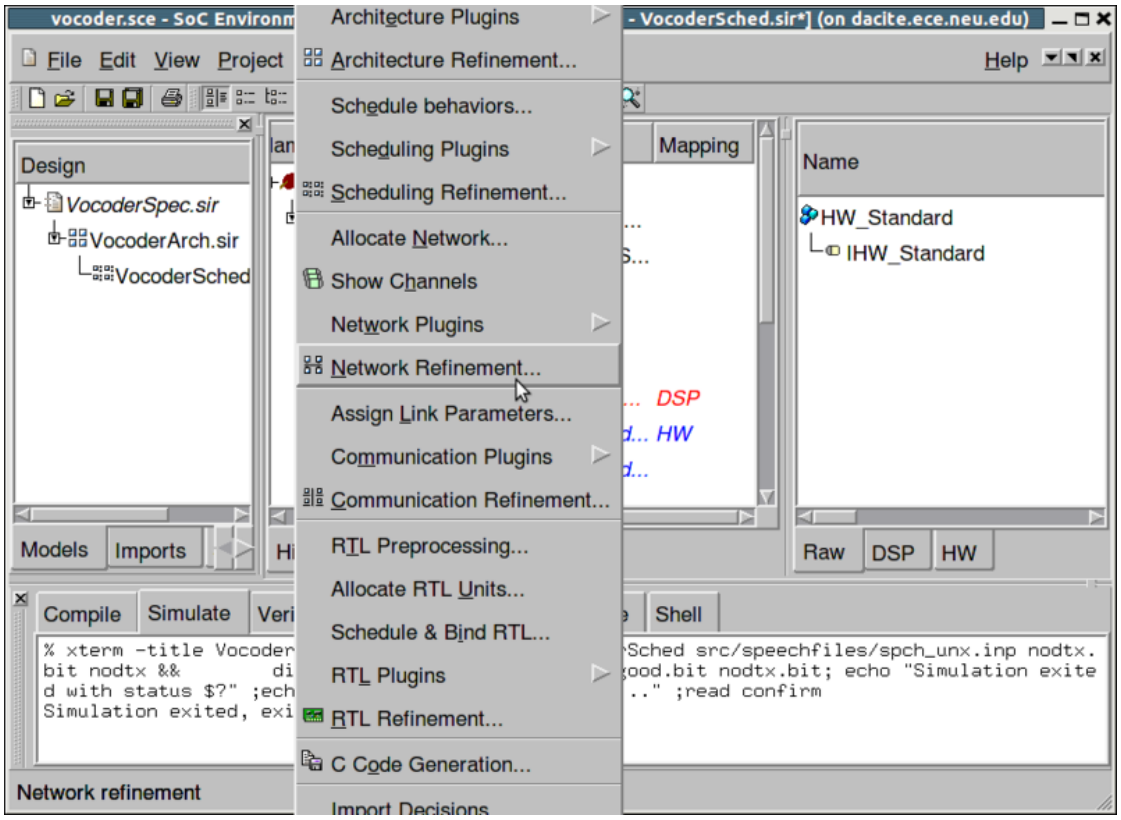
Expand the design hierarchy window and scroll to the right to find a new column entry Bus.

3.4.2.1. Map channels to buses (cont'd)



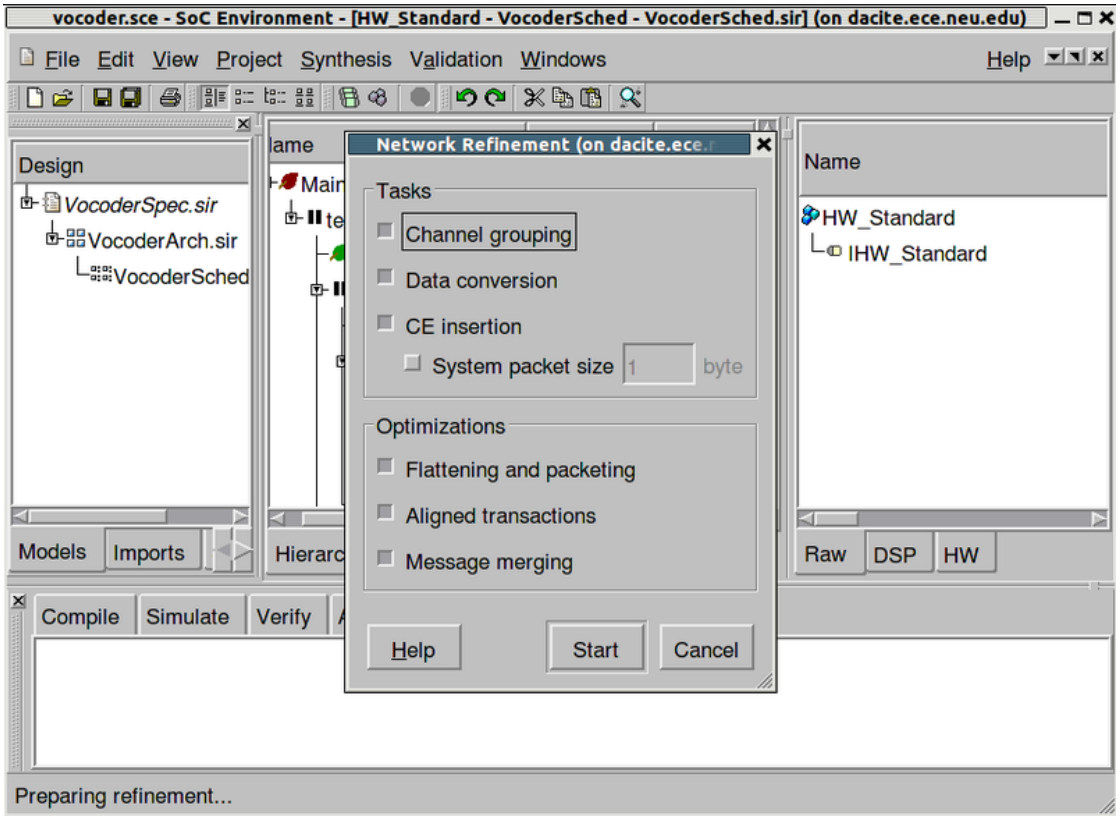
Like component mapping, bus mapping may be done by assigning variable channels to buses. However, to speed things, we may assign the top level component to our system bus. Since we have only one system bus, all the channels will be mapped to it. This is done by left clicking in the row for the "Coder" behavior under the bus column. Select the default "Bus0" and press RETURN.

3.4.3. Generate communication model



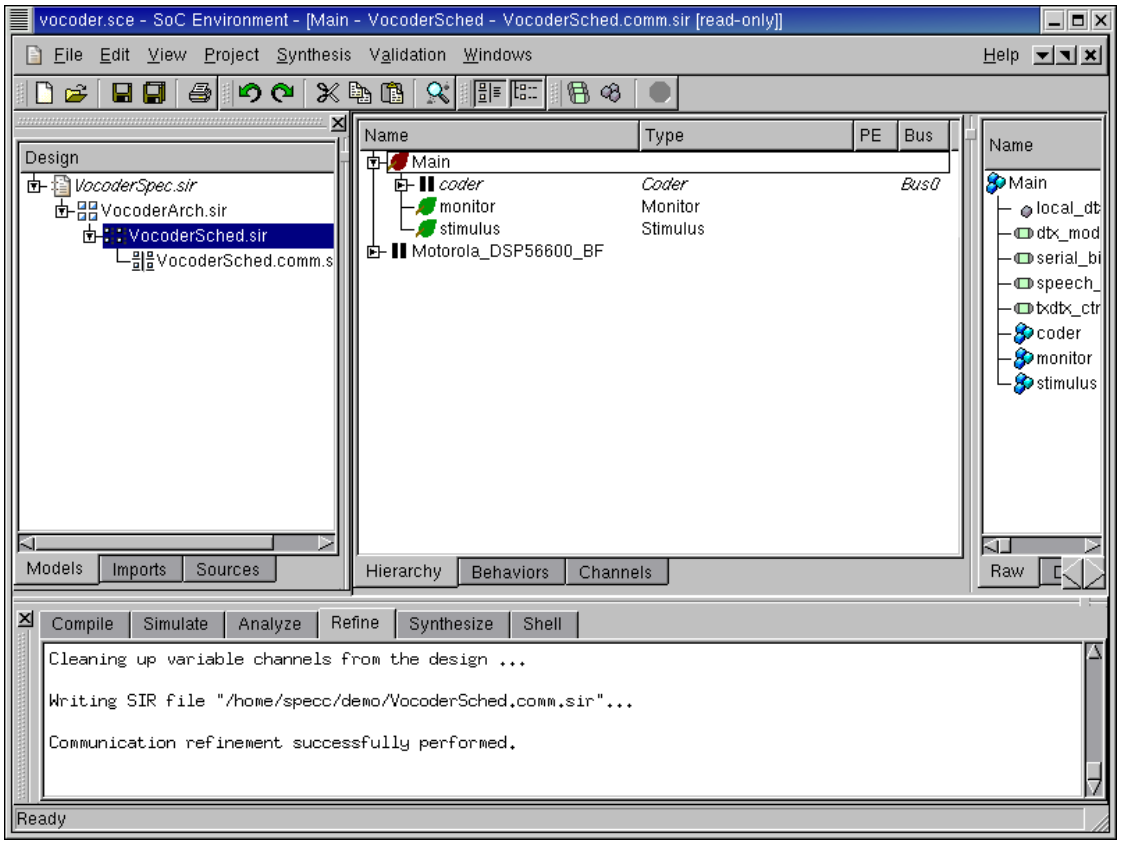
Now that we have completed bus allocation and mapping, we may proceed with network refinement. Like architecture refinement, this process automatically generates a new model that reflects our desired bus architecture. To invoke the network refinement tool, select Synthesis⟶Network Refinement from the menu bar.

3.4.3.1. Generate communication model (cont'd)



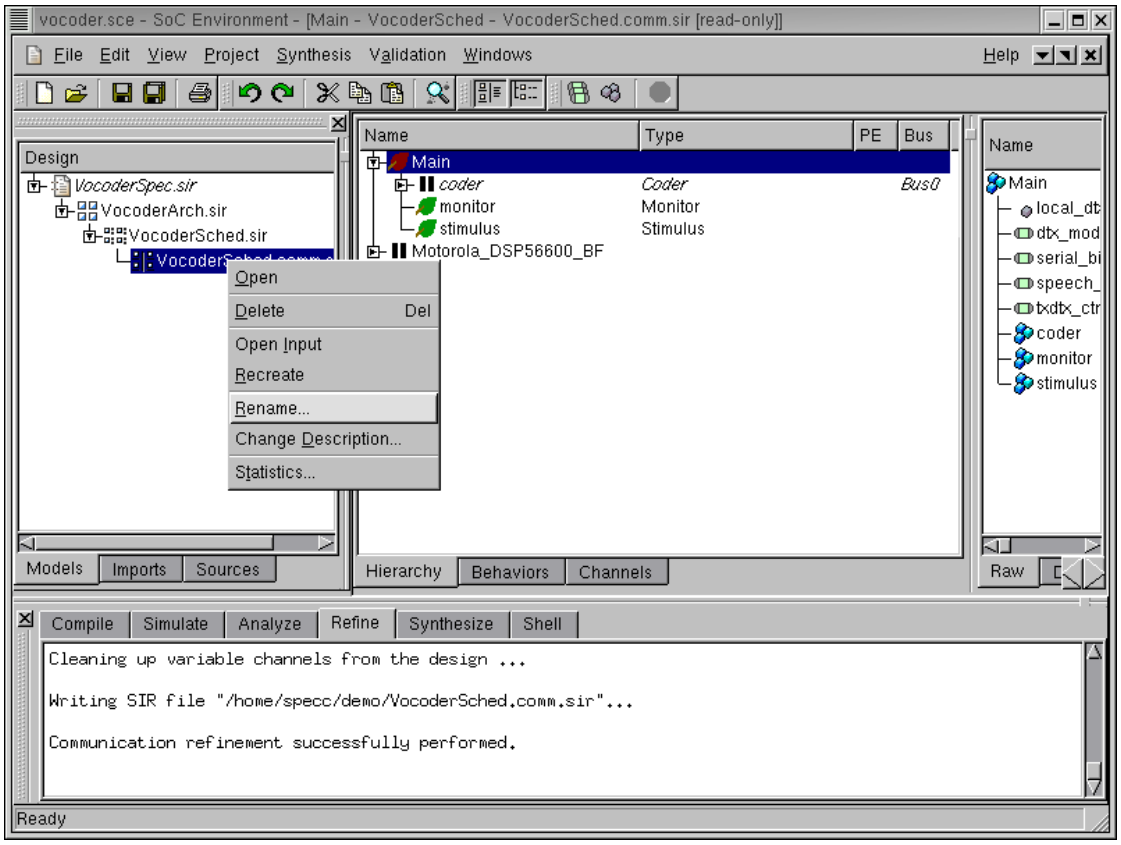
A new window pops up giving the user the option to perform various stages of the refinement. The user may choose to partially refine the model without actually inserting the bus, and only selecting the channel refinement phase. This way, he can play around with different channel partitions. Likewise, the user might want to play around with different bus protocols while avoiding "Inlining" them into components. This way he can plug and play with different protocols before generating the final inlined model. By default all the stages are performed to produce the final communication refinement. Since we have only one bus, and hence a default mapping, we opt for all three stages and left click on Start to proceed.

3.4.3.2. Generate communication model (cont'd)



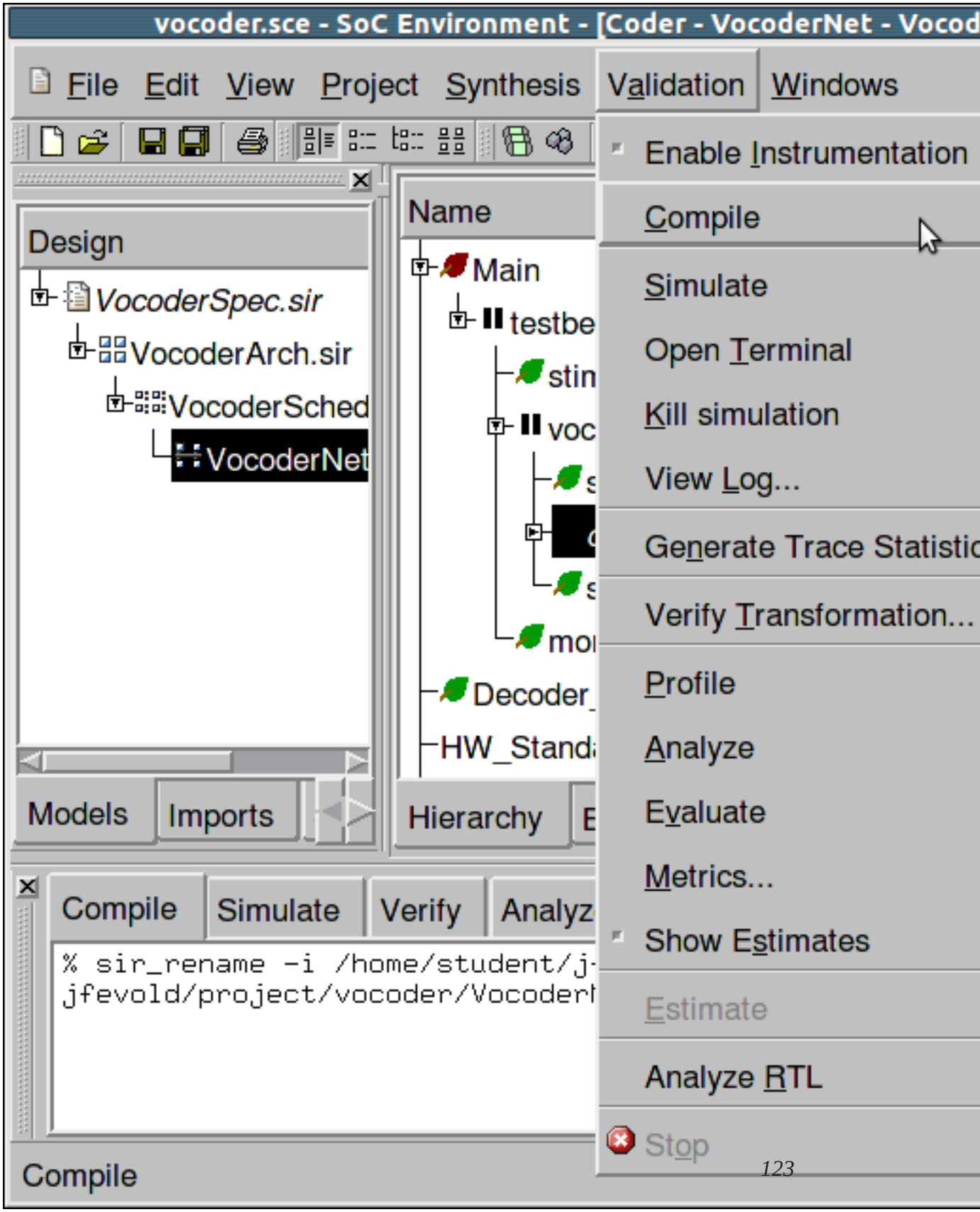
During communication refinement, note the various tasks being performed by the tool in the logging window. The tool reads in channel partitions, groups them together, imports selected busses and their protocols, implements variable channel communication on busses and finally inlines the bus drivers into respective components. Once communication refinement has finished, a new model is added in the project manager window. It is named "VocoderArch.comm.sir". Also note that we have a new design management window on the right side in the GUI.

3.4.3.3. Generate communication model (cont'd)



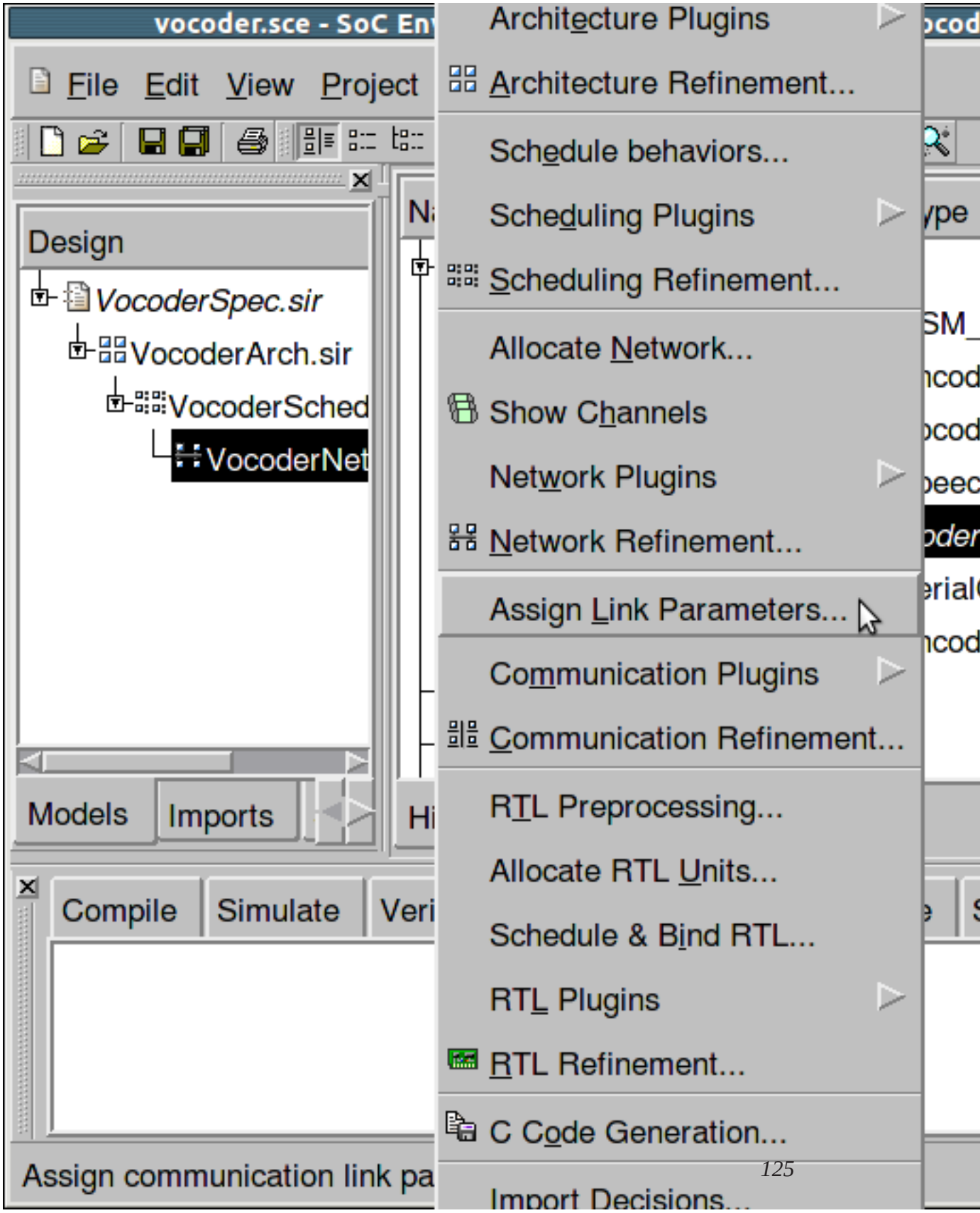
We now need to give our newly created communication model a reasonable name. To do this, right click on "VocoderArch.comm.sir" in the project manager window and select Rename from the pop-up menu. Now rename the model to "VocoderNet.sir".

3.4.3.4. Generate communication model (cont'd)



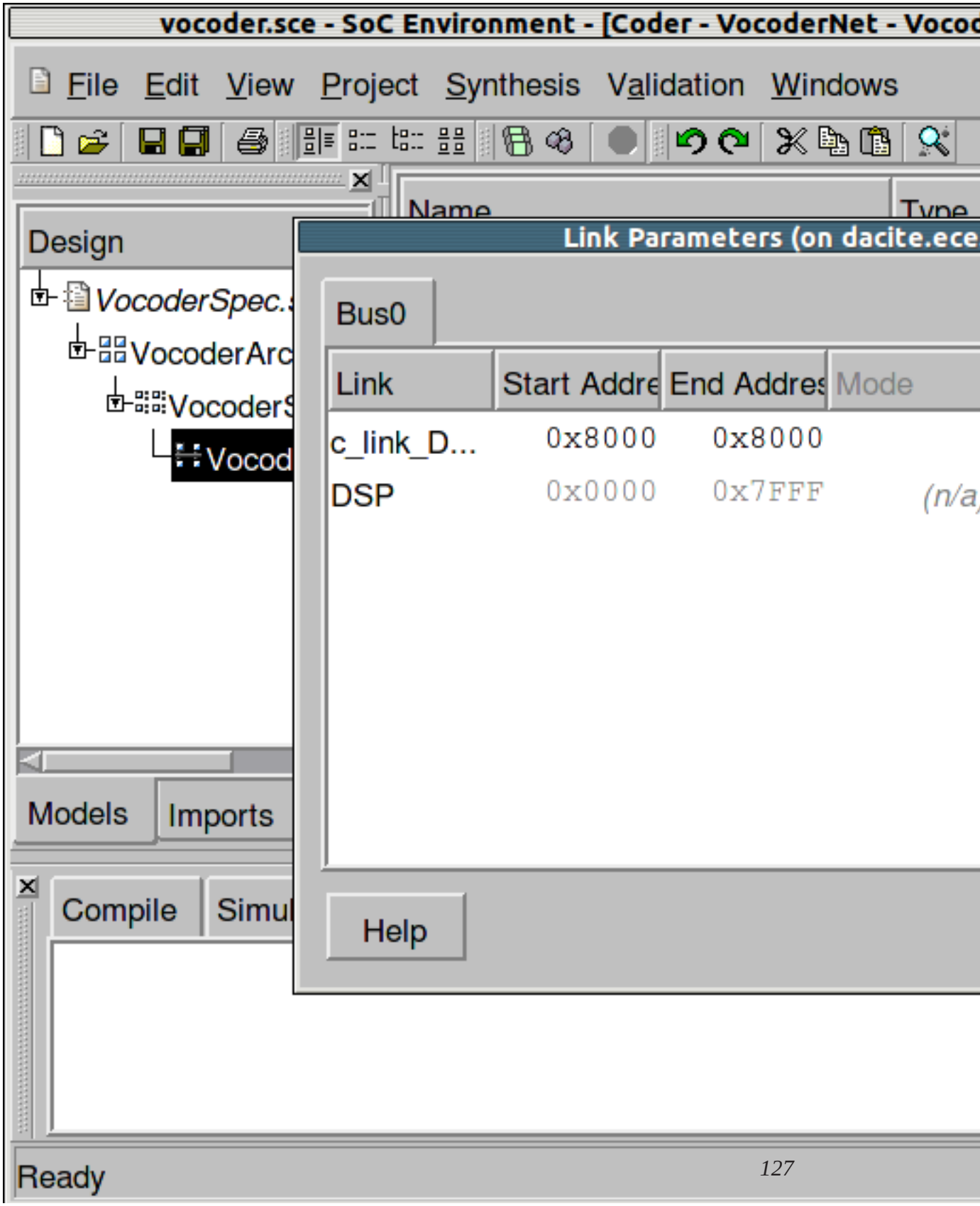
Simulate the resulting network model of the design by selecting Validation⟶Compile. Upon completion, select Validation⟶Simulate to validate its correctness.

3.4.3.5. Generate communication model (cont'd)



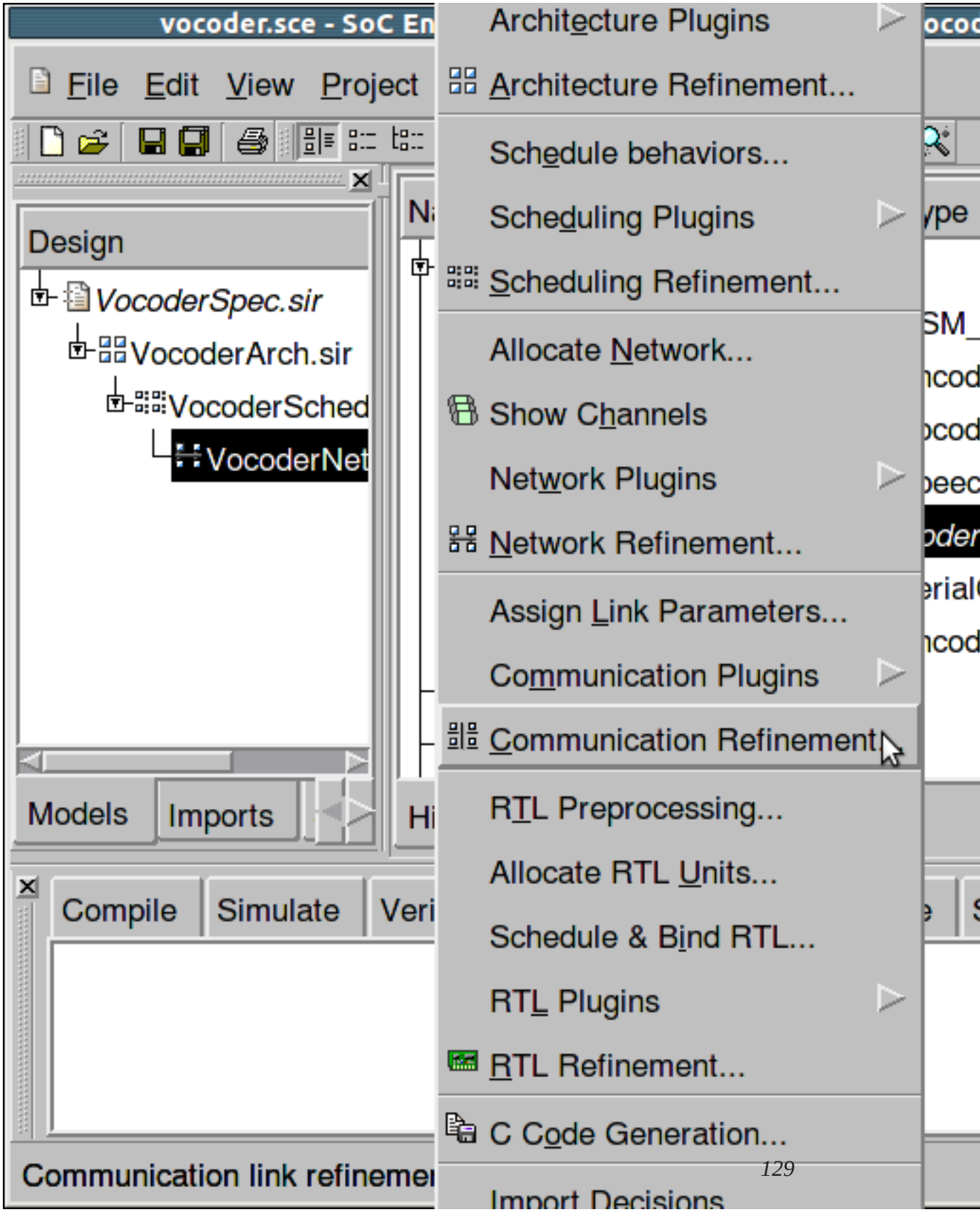
After network refinement, synthesis of communication in each bus segment of the system has to be done to complete the communication design process. For this, we start by setting parameters for all comunication links in the system. Select Synthesis⟶Assign Link Parameters.

3.4.3.6. Generate communication model (cont'd)



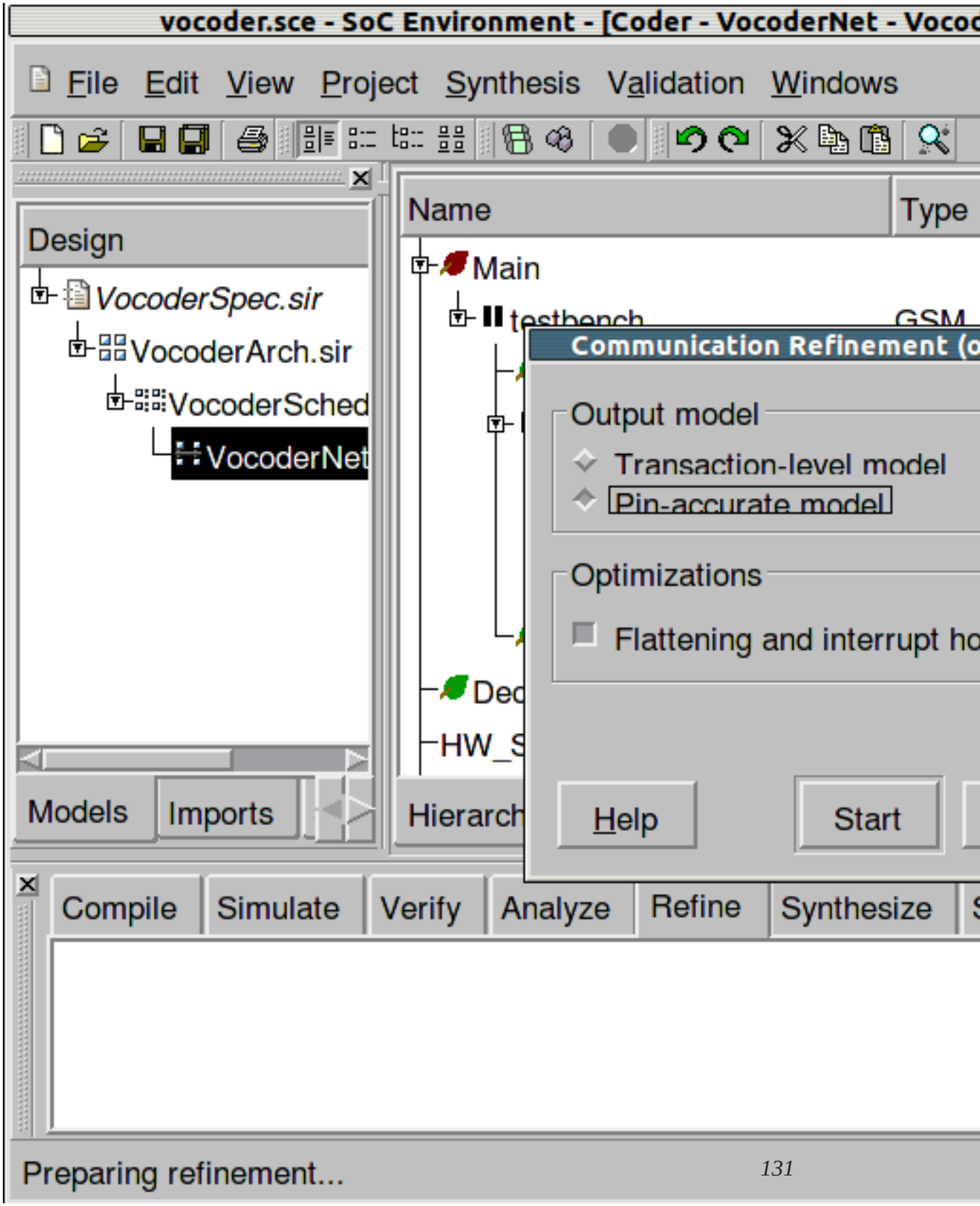
The Link Parameter dialog come up with one tab for each bus in the system. In the Link Parameters dialog, address maps and synchronization mechanisms for every channel and every PE interface on the busses have to be defined. On "Bus0," select the Start address of "c_link_DSP__HW" to be a value between 0x8000-0xFFFF and the Interrupt to be "MasterIntA" from the drop down menu. Finally, click OK to accept and confirm the bus parameter definition.

3.4.3.7. Generate communication model (cont'd)



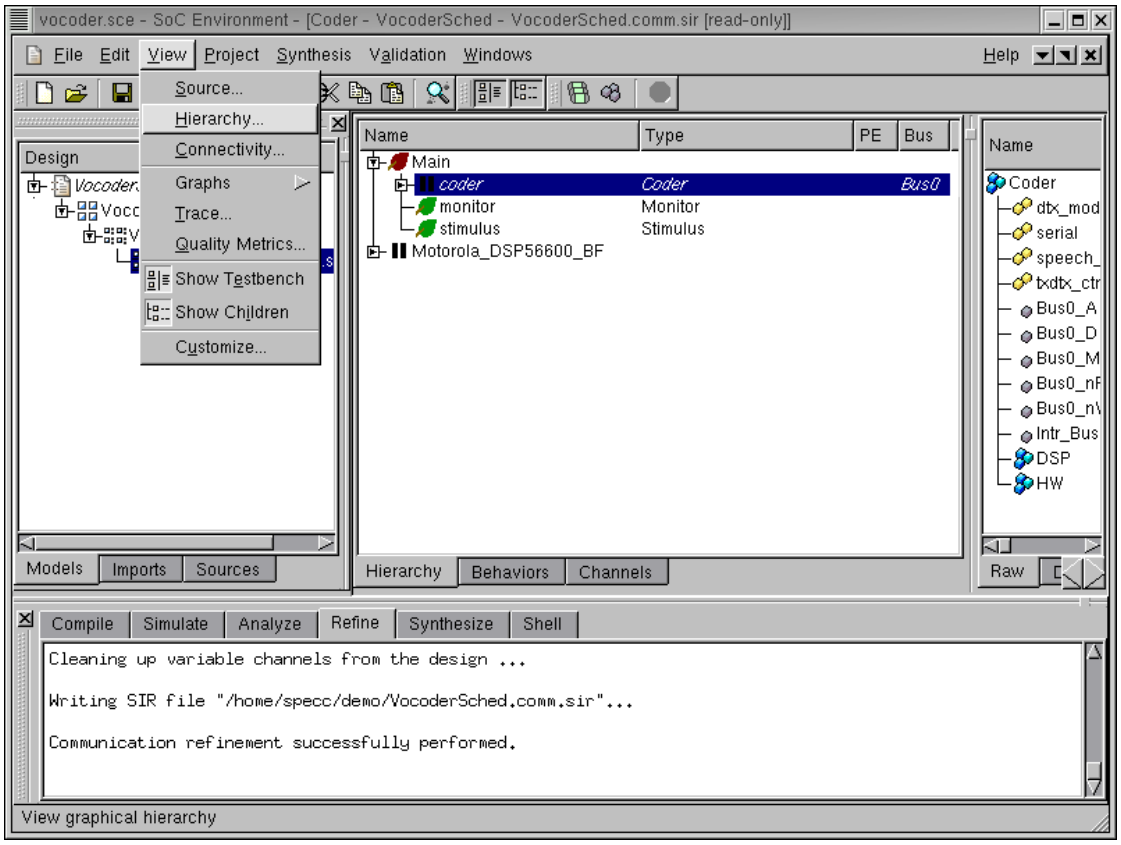
Select Synthesis⟶Communication Refinement to invoke the final communication refinement tool.

3.4.3.8. Generate communication model (cont'd)



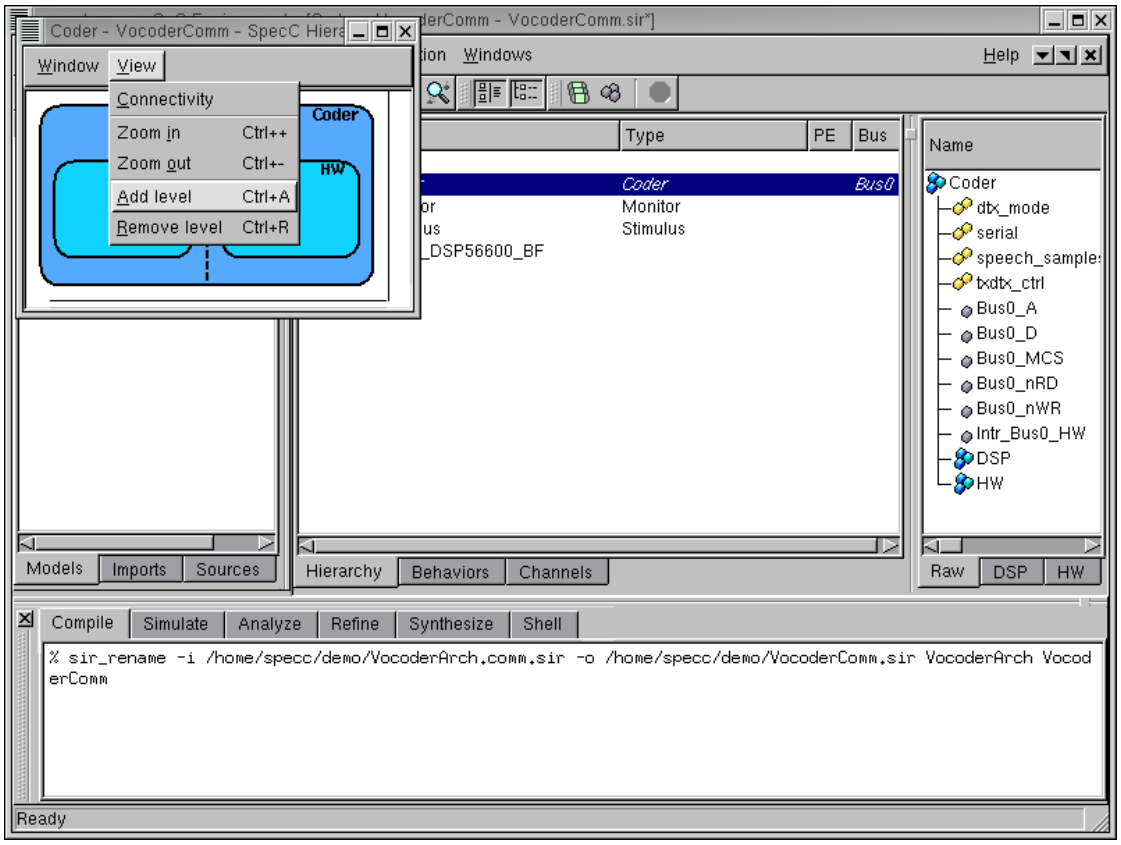
The dialog that pops up provides the option either to generate a Pin Accurate Model or a Transaction Level Model of the design. Accept the default option of generating a PAM and click Start to proceed.

3.4.4. Browse communication model



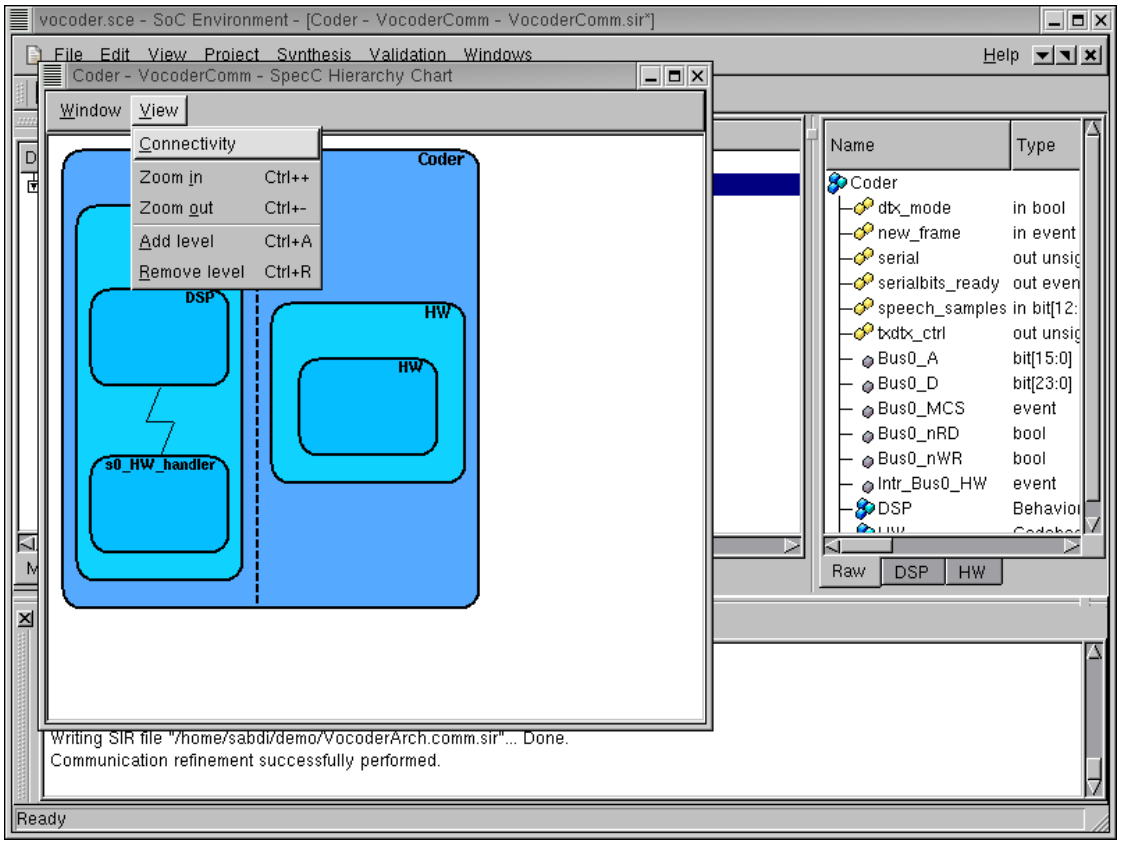
Like we did after architecture refinement, we browse through the communication model generated by the refinement tool. We have to first check whether it is semantically and structurally representing a model as described in our SoC methodology. To observe the model transformations produced by communication refinement, we need a graphical view of the model. This is done by left clicking to choose the "Coder" behavior in the design hierarchy window and selecting View⟶Chart from the menu bar.

3.4.4.1. Browse communication model (cont'd)



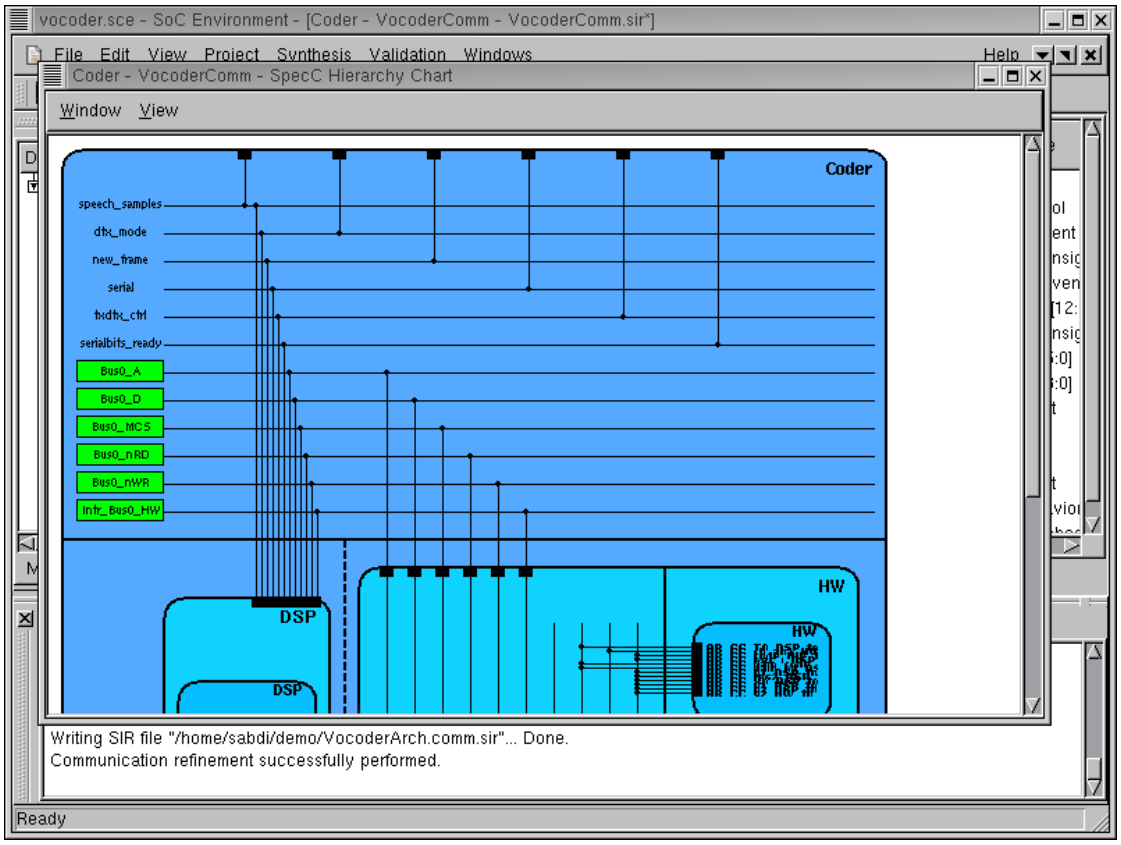
A new window pops up showing the model with DSP and HW components. We have to observe the bus controllers generated during refinement and the added details to the model. Hence, we select View⟶Add level from the menu bar to view the model with greater detail.

3.4.4.2. Browse communication model (cont'd)



In the next level of detail, we can now see the interrupt handler "s0_HW_handler" behavior added in the master to serve interrupts from the HW slave. To view the actual wire connections of the system bus, enlarge window and select View⟶Connectivity from the menu bar.

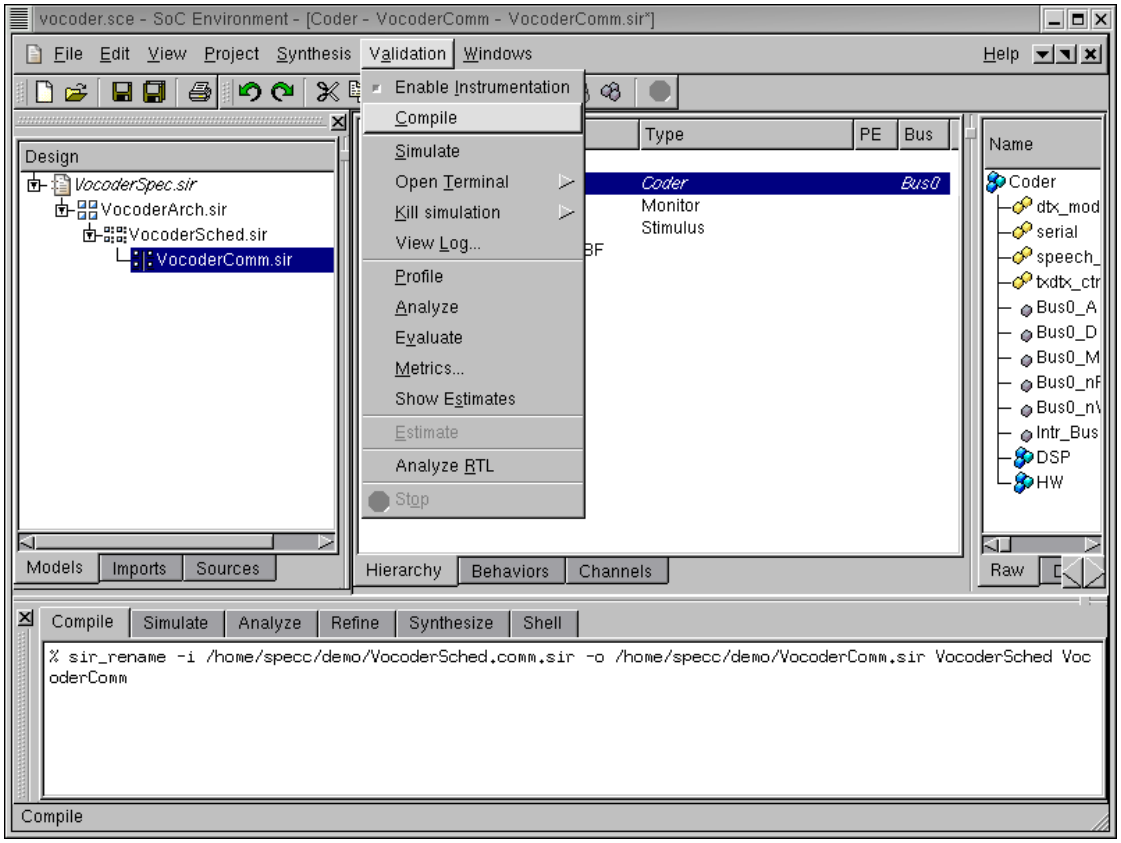
3.4.4.3. Browse communication model (cont'd)



The wire level detail of the connection between components can now be seen in the window. Note that the system bus wires are distinguished by green boxes. Hence we see that the bus is introduced in the design and the individual components are connected with the bus instead of the abstract variable channels. On observing the hierarchical view further, we can see the drivers in each components. These drivers take the original variables and implement the high-level send/receive methods using the bus protocol.

We have thus seen that the structure of communication model follows the semantics of the model explained in our methodology. We may complete the browsing session by selecting Window⟶Close from the menu bar.

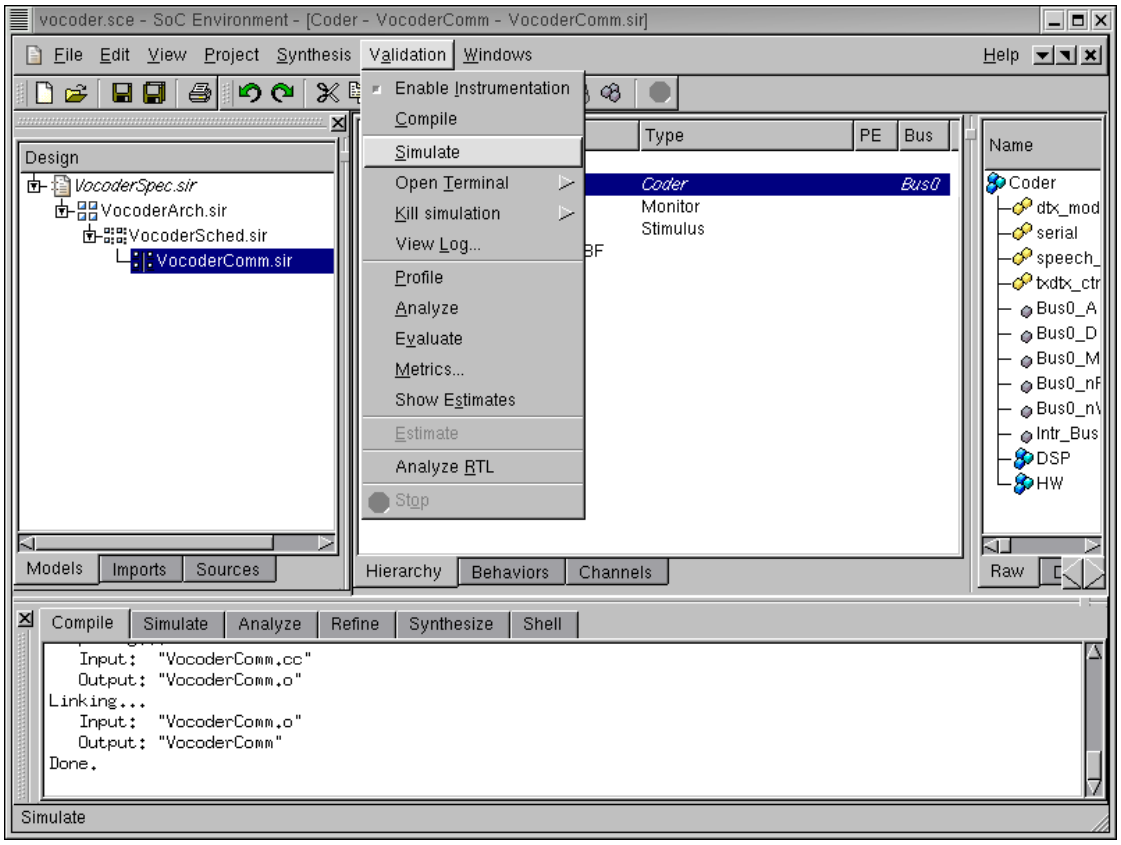
3.4.5. Simulate communication model (optional)



This section shows the simulation of the generated communication model. If the reader is not interested, she or he can skip this section and go directly to Section 3.5 *Summary* (page 140).

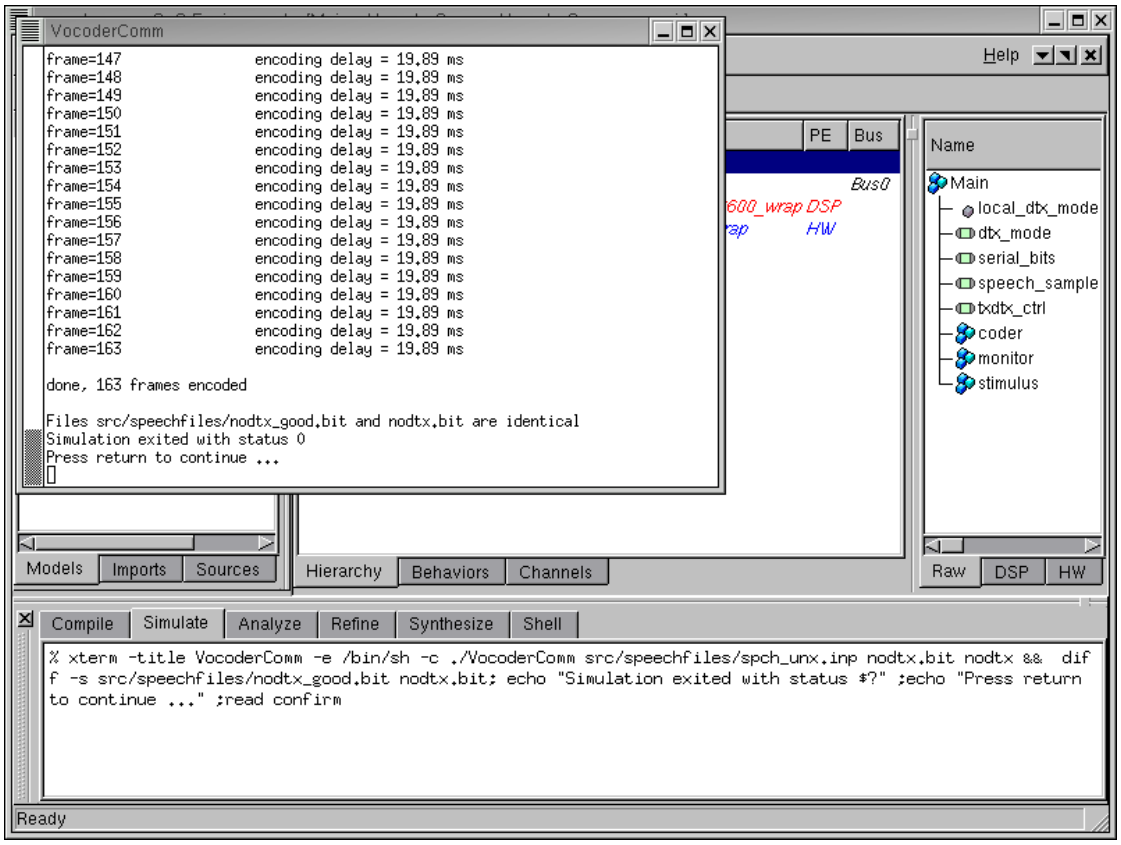
As a direct analogy to the validation of the architecture model, we have a step for validating the communication model. The newly generated model has already been verified to adhere to our notion of a typical communication model. We must now verify that the communication model generated after the refinement process is functionally correct or not. Toward this end, the model is first compiled. This is done by selecting Validation⟶Compile from the menu bar.

3.4.5.1. Simulate communication model (optional) (cont'd)



The model should compile without errors and this may be observed in the logging window. Once the model has successfully compiled, we must proceed to simulate it. This is done by selecting Validation⟶Simulate from the menu bar.

3.4.5.2. Simulate communication model (optional) (cont'd)



An xterm now pops up showing the simulation in progress. Note that simulation is considerably slower for the communication model than for the architecture and communication model. This is because of the greater detail and structure added during the refinement process. Also, it may be noted that the execution time for encoding each frame goes up to 19.89 ms from 19.77 ms, which we had for the model before communication synthesis. This is because communication synthesis replaced the abstract untimed transactions with detailed, timed bus protocols, which introduces non-zero communication delay. However, the execution time is still well within the 20 ms constraint for encoding each frame.

With the completion of correct model simulation, we are done with the phase of communication synthesis. Our new model now has two components connected by a system bus. The model is now ready for implementation synthesis.

3.5. Summary

In this chapter, we covered the system level design phase of our methodology. With the rise in level of abstraction in system specification, it is no longer feasible to start designs at cycle accurate level. Instead, the specification should be gradually refined to derive a cycle accurate model. We saw three major steps in the system level design and synthesis process.

Architecture refinement took in the system specification model as input. Based on the profile of the specification, we chose the appropriate components to implement the desired system. We also delved into design space exploration by seeking a purely software solution. When the software solution turned out to be infeasible, we added a HW component to meet the real-time constraint of the design. We also demonstrated the power of automatic refinement to quickly come up with models and evaluate them, thereby greatly enhancing design space exploration. In the future, we will look at how to automate the decision making process, so that the tool can propose an optimal system architecture based on system constraints and available components.

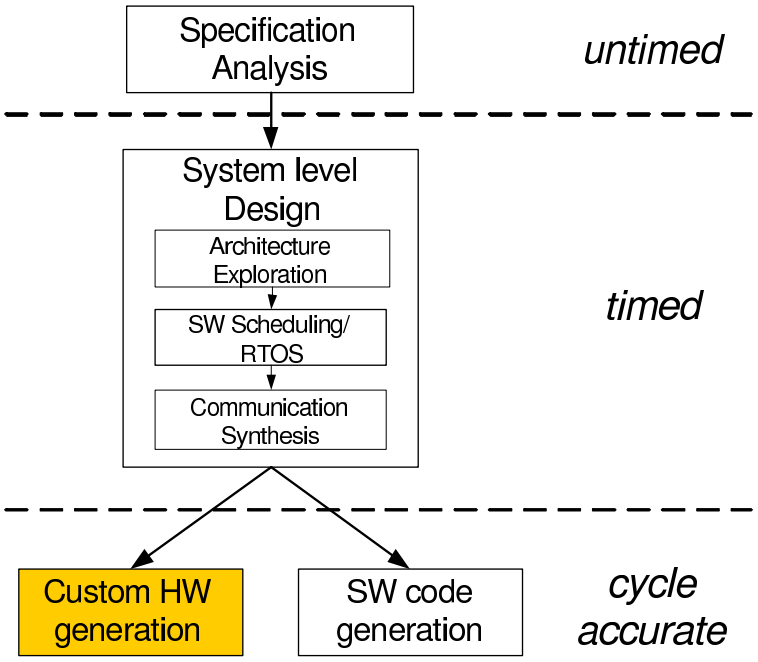
Architecture refinement was followed by software scheduling and the RTOS insertion step. Although, for this demo, we did not need to insert any RTOS, it is a feature available in SCE. It allows for inclusion of useful task scheduling algorithms for dynamic scheduling. We also provide for static scheduling of tasks on both HW and SW.

The final major step of system level design is communication synthesis. We showed how the designer can use the database of a variety of bus models to construct a communication architecture for the design. Once the communication architecture is complete, the designers can assign abstract data transfers to a communication route in the architecture. Using automatic refinement in SCE, we showed how the designer could quickly produce a bus functional communication model and see if it fits the system requirements. This bus functional model serves as an input to the tasks of custom HW generation and SW code generation, which are described in the next two chapter. In the future, we would like to enhance the capabilities of our tool to perform automatic communication synthesis, whereby the tool can generate a good communication architecture and still meet system specification constraints.

Chapter 4. Custom Hardware Design

4.1. Overview

Figure 4-1. Custom hardware generation using SCE



In this chapter, we look at custom HW generation step as highlighted in Figure 4-1. The bus functional model derived from the system level design phase must now be used to generate custom hardware for HW components. In this phase of RTL synthesis, our goal is to generate an RTL model that can be fed into industry standard synthesis tools. In this chapter, we will deal exclusively with behaviors mapped to HW components and show how a cycle accurate model is derived from a bus functional one.

First, super finite state machine with data (SFSMD) will be generated from the communication model. Each super state in SFSMD corresponds to a basic block in communication model and will have only data flow information. The control flow information will be described among super states of the behavior. Super states in SFSMD will be split into multiple states during RTL synthesis.

Second, the RTL units for the custom hardware are allocated. To get some information like number of operations, number of variables and number of data transfers in the SFSMD for the RTL allocation, the designer has to run RTL analysis tool.

Third, scheduling and binding is done by designer or by tools. The scheduling and binding information will be inserted into the SFSMD model.

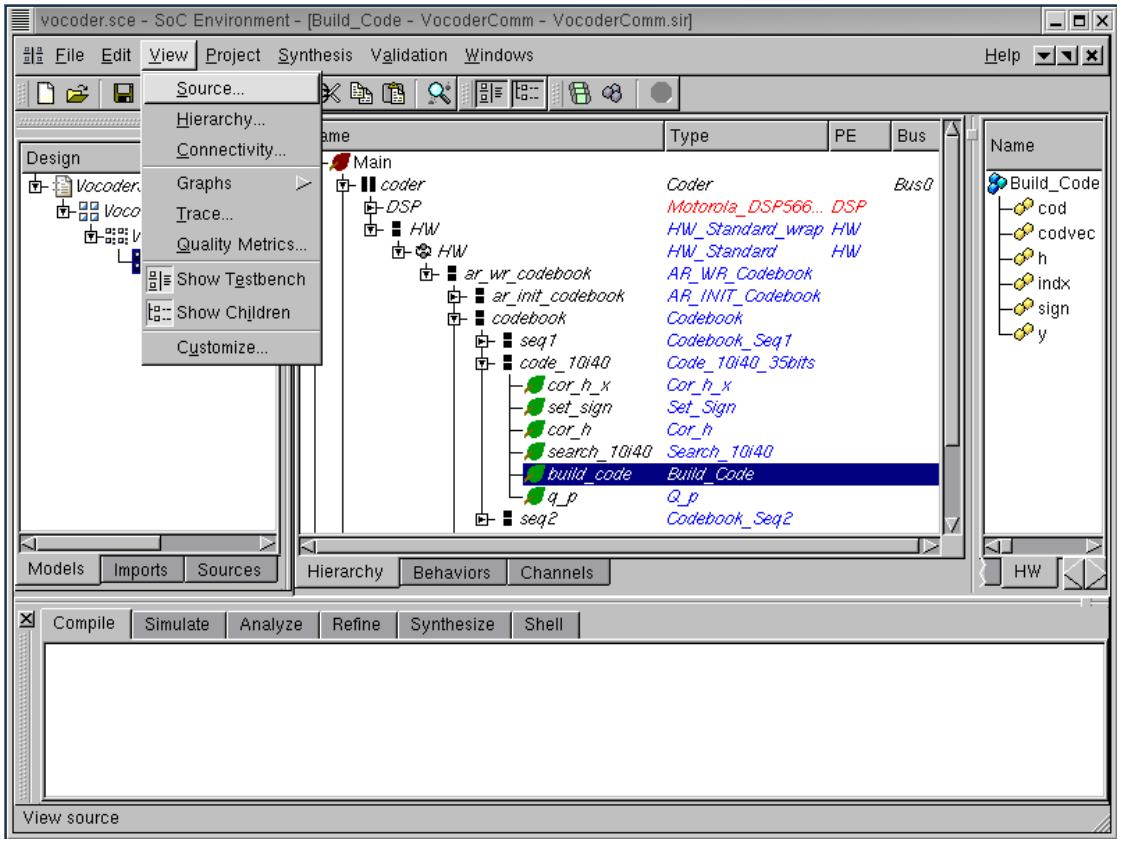
Finally, the SFSMD model with scheduling and binding information is refined into a cycle-accurate FSMD model by RTL refinement tool. The refinement tool will also gererate a cycle accurate model in hardware description languages like Verilog and Handel-C. The cycle accurate model in Verilog HDL can be used as input to commercial logic synthesis tools like Synopsys Design Compiler. We also generate the cycle-accurate model in Handel-C which can be fed into Celoxica Design Kit.

4.2. RTL Preprocessing

In our design methodology, RTL design is modeled by Finite State Machine with Data (FSMD) which finite state machine model with assignment statements added to each states. The FSMD can completely specify the behavior of an arbitrary RTL design.

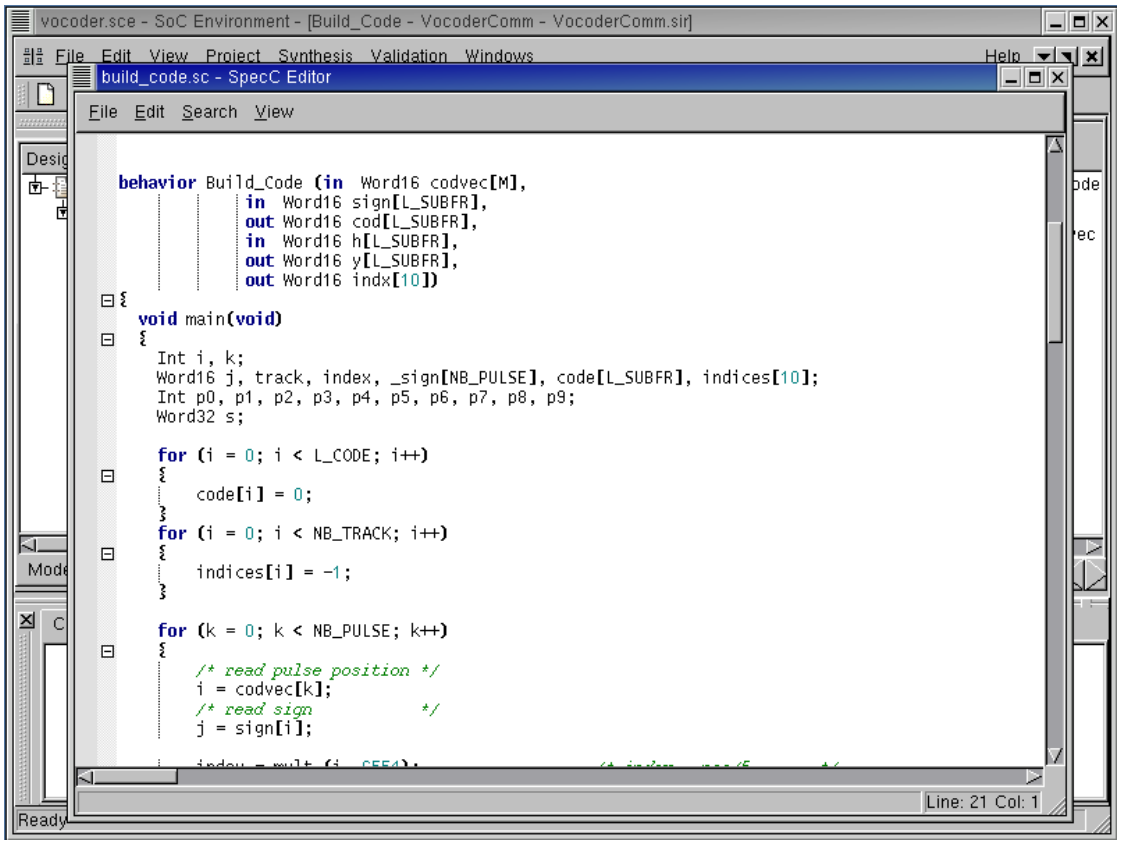
In this tutorial, we use an intermediate representation, super finite state machine with data (SFSMD), where each state may take more than one cycle to execute. The SFSMD will be automatically refined into cycle-accurate FSMD after RTL scheduling and binding.

4.2.1. View behavioral input model



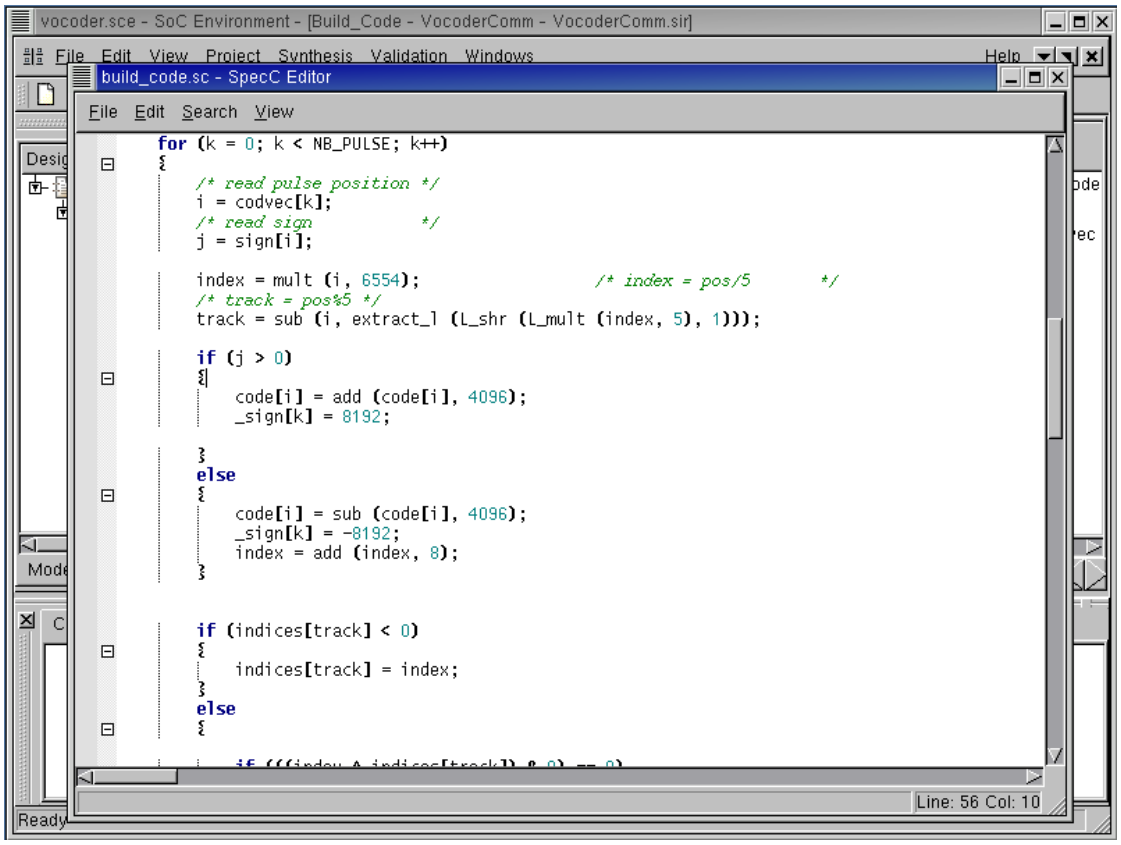
Before we show how to generate SFSMD, we take a look at how input model of custom hardware design. Select the behavior "Build_Code" by left clicking on it. We can take a look at the behavioral input model by selecting View⟶Source from the menu bar.

4.2.1.1. View behavioral input model (cont'd)



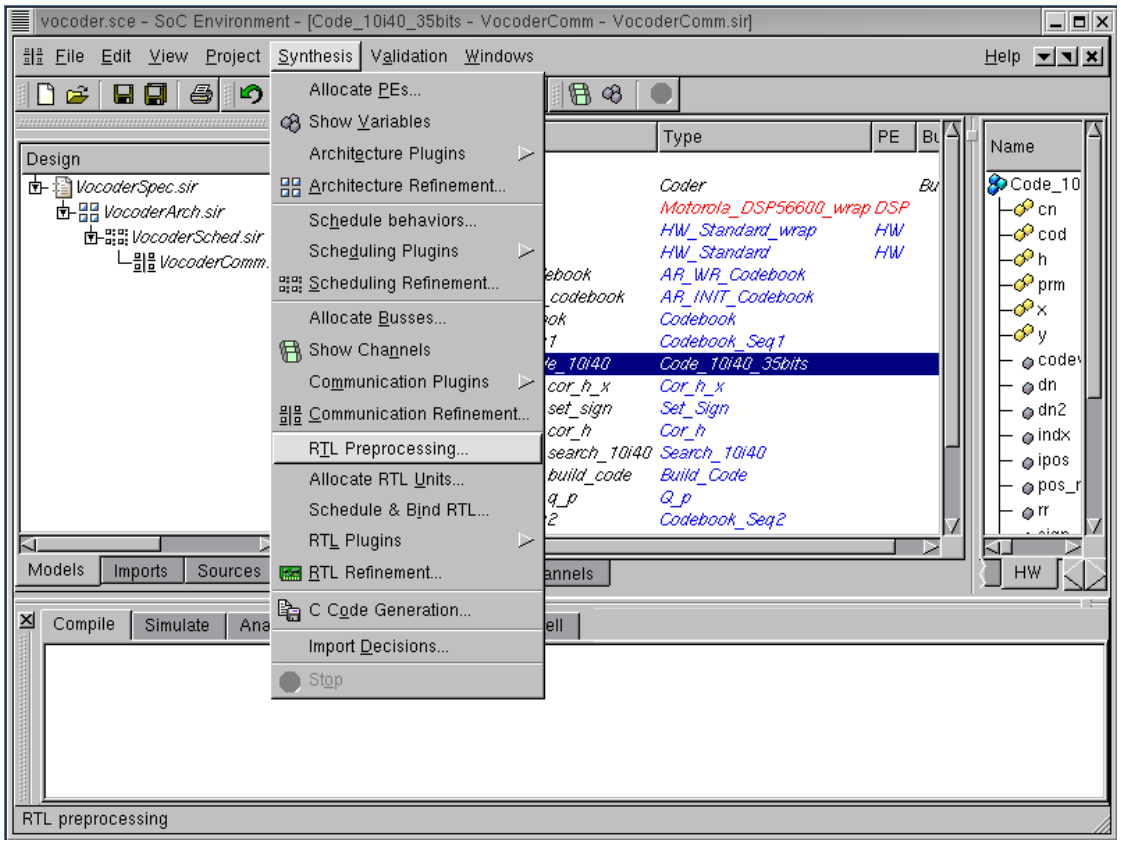
The SpecC Editor window pops up showing the source code for behavior "Build_Code".

4.2.1.2. View behavioral input model (cont'd)



Scrolling down the window, we can see that the behavior code has loops and conditional branch constructs. Therefore, our RTL synthesis tool has to handle these constructs. Close the SpecC Editor window by selecting File⟶Close from its menu bar.

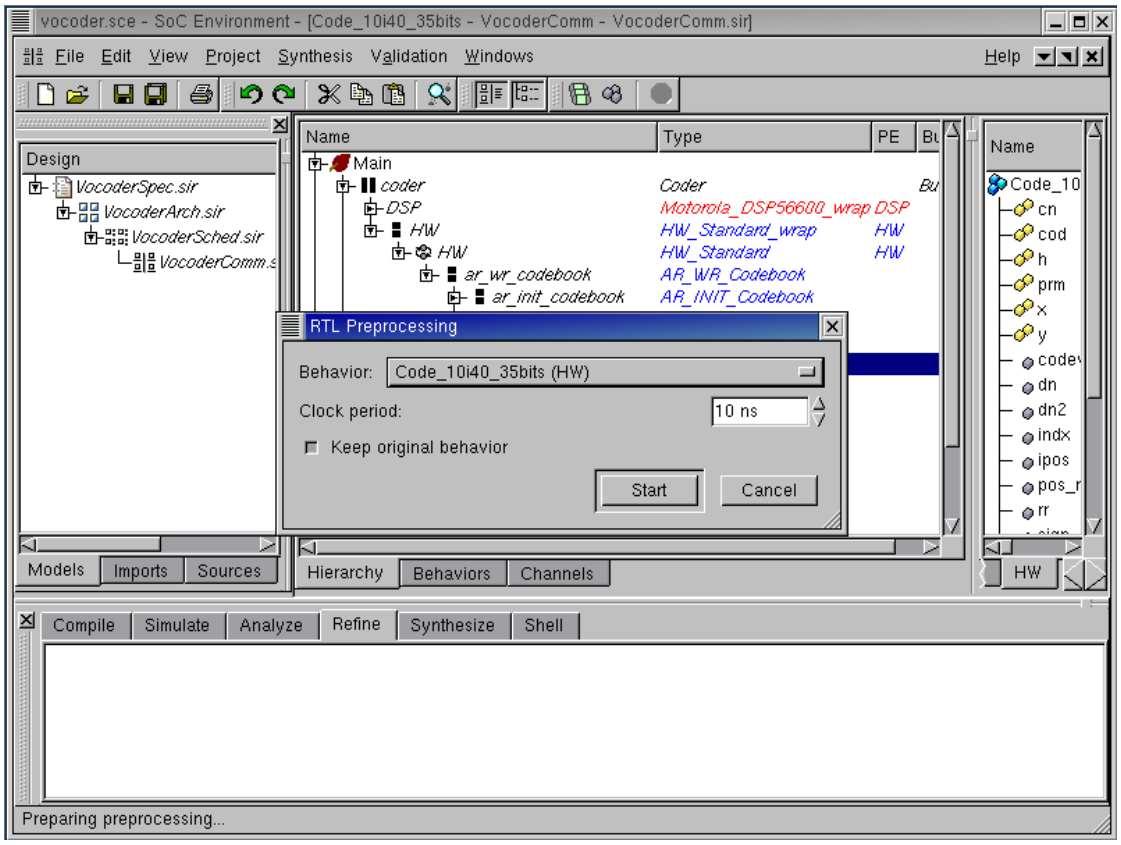
4.2.2. Generate SFSMD model



Now, we will show how to generate super finite state machine with data (SFSMD). To demonstrate the features of the our custom hardware synthesis tool, we will use a particular behavior called "Code_10i40_35bits". Browse the hierarchy in the design hierarchy window and select behavior "Code_10i40_35bits". We will be demonstrating RTL design exploration with this behavior in the rest of the chapter.

In the SCE, the step of generating the SFSMD from the behavioral input model is called RTL preprocessing, which is necessary for RTL synthesis. RTL preprocessing can be invoked by selecting Synthesis⟶RTL Preprocessing from the menu bar.

4.2.2.1. Generate SFSMD model (cont'd)

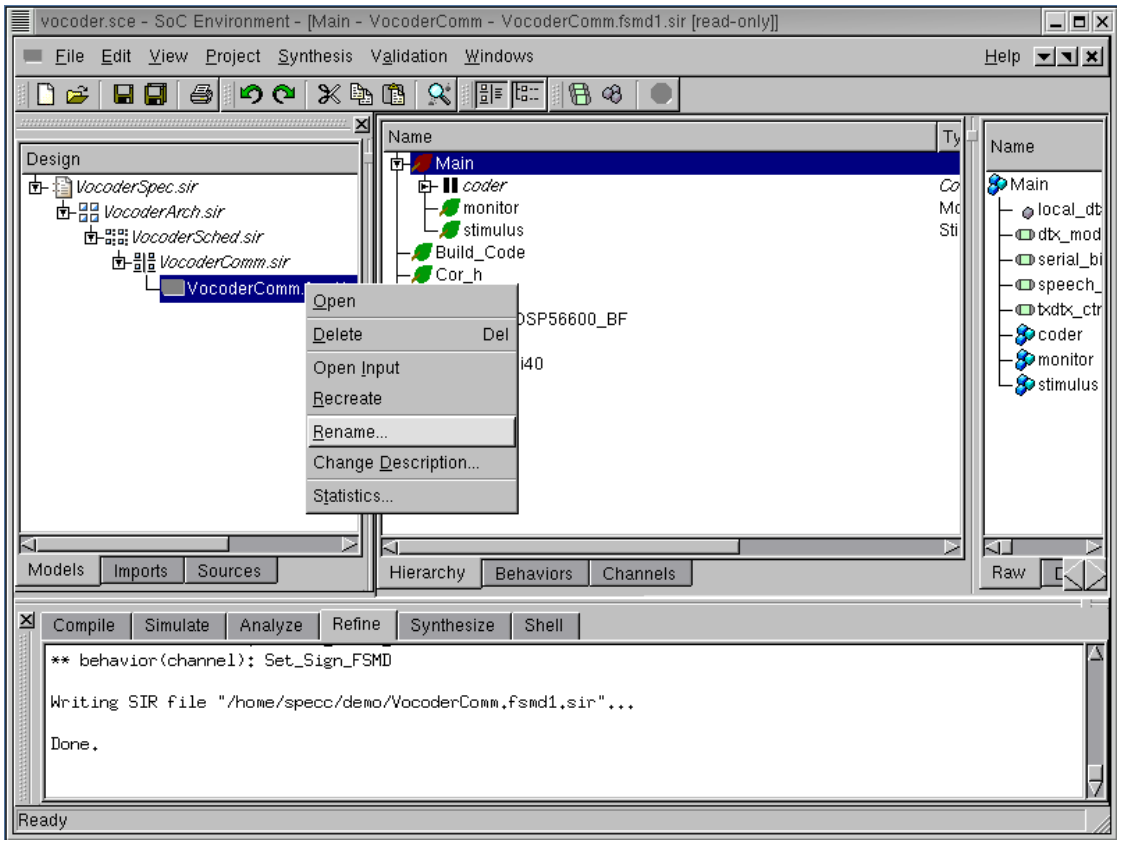


An RTL Preprocessing dialog box pops up for selecting the behavior and its clock period. Select "Code_10i40_35bits" as the behavior to be preprocessed and leave the default clock period of the behavior as 10 ns. Note that the clock period here is used only for generating a simulatable FSMD construct in SpecC. It does not mean that each state in the SFSMD model will eventually take 10 ns to execute.

In the dialog box, the option Keep original behavior means that the original behavior definitions for "Code_10i40_35bits" and its sub-behaviors will be preserved in the model. Their instances, however, will be replaced by the generated SFSMD behavior instances in the hierarchy.

Now click Start to begin preprocessing.

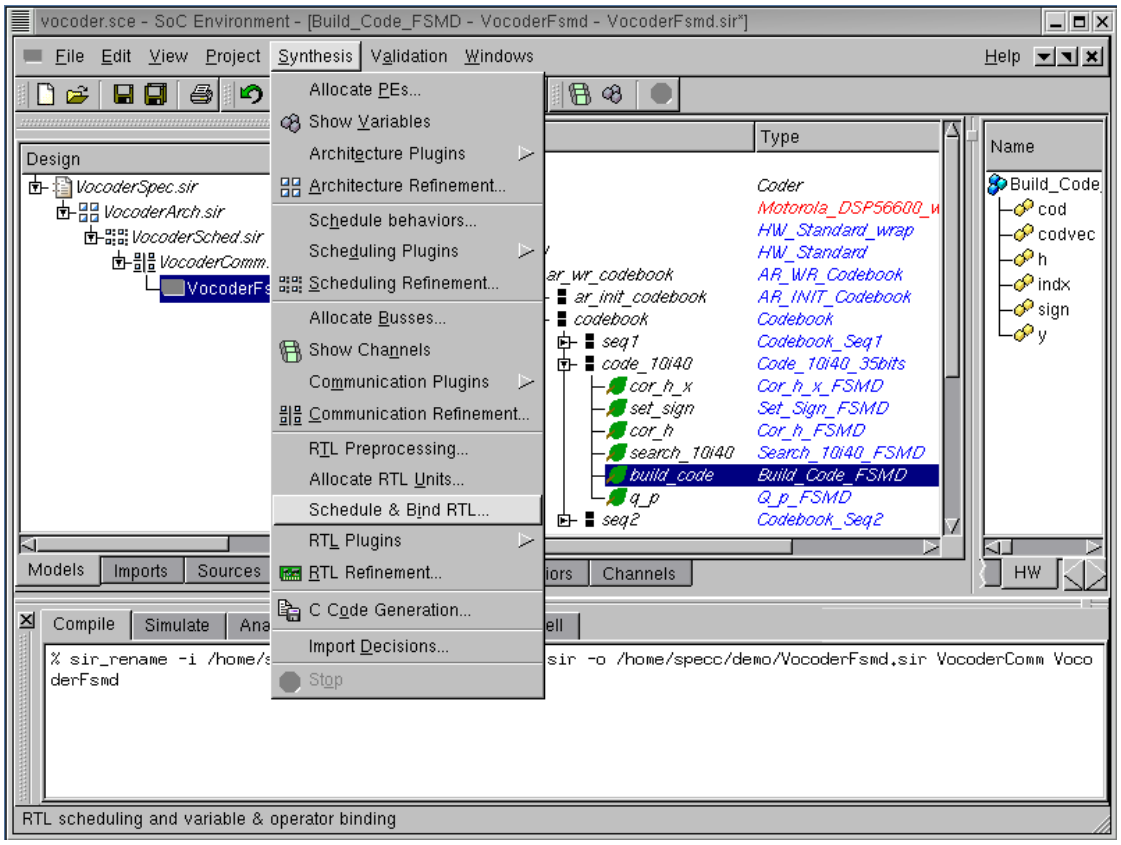
4.2.2.2. Generate SFSMD model (cont'd)



Note that RTL preprocessing step generates new SFSMDs for 6 sub-behaviors in the behavior "Code_10i40_35bits", as seen on the logging window. Also note that a new model "VocoderComm.fsmd.sir" is added in the project manager window. This new model contains SFSMD behaviors mapped to HW component, which can be seen in the design hierarchy tree.

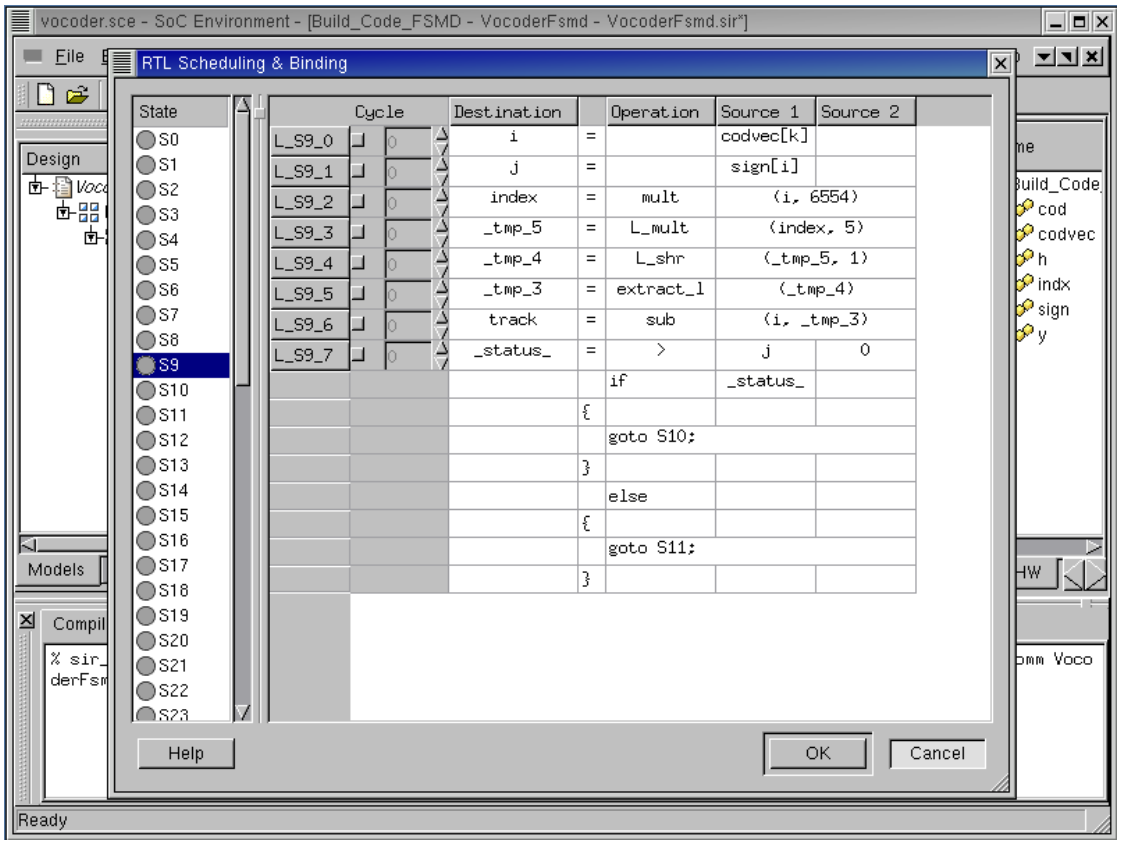
Again, we must give our new model a suitable name. We can do this by right clicking on "VocoderComm.fsmd.sir" and selecting Rename from the pop up menu. Rename the model to "VocoderFsmd.sir".

4.2.3. Browse SFSMD model



Select the behavior "Build_Code_FSMD" from the hierarchy by left clicking on it. The generated SFSMD leaf behaviors may be viewed by selecting Synthesis⟶Schedule & Bind RTL from the menu bar.

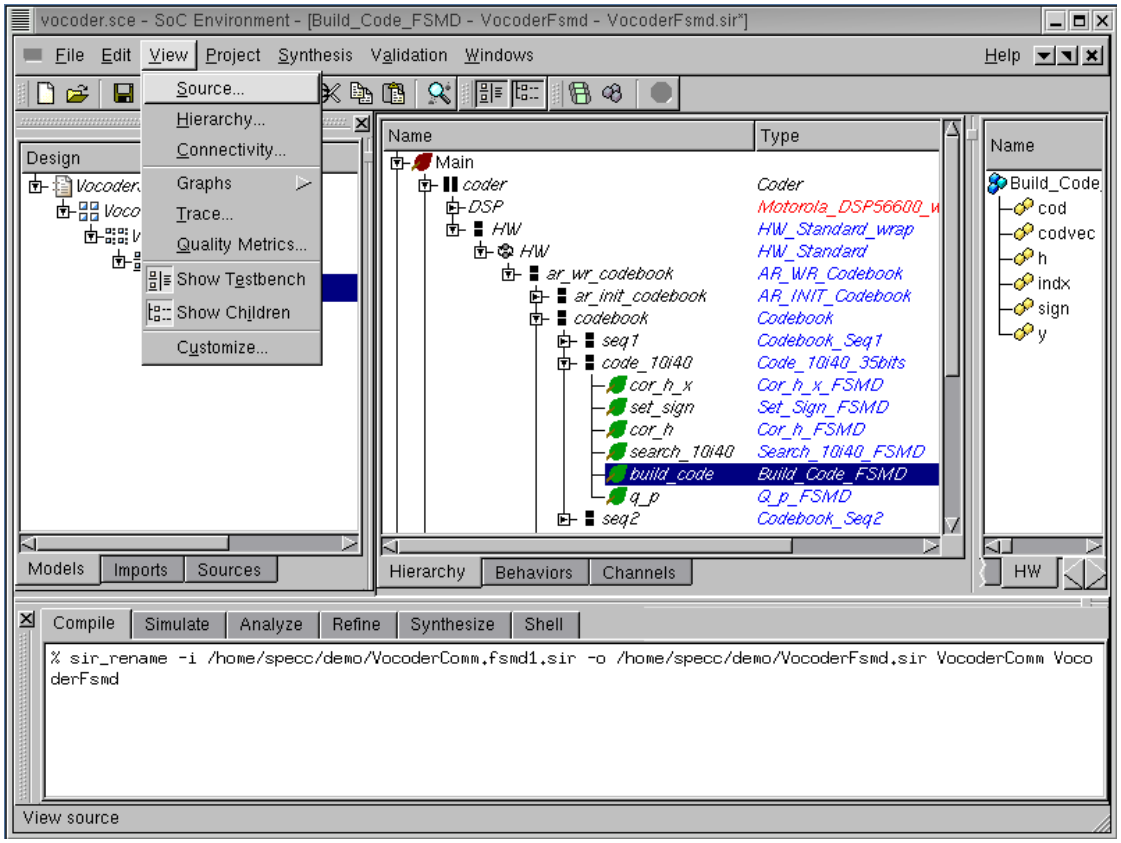
4.2.3.1. Browse SFSMD model (cont'd)



The RTL Scheduling & Binding window pops up showing all the states in the behavior "Build_Code_FSMD". It also shows all statements for the selected state in the right-most column. We can go inside each state by clicking on the corresponding circle in the left-most column. In this screen shot, state S9 is selected. We can see all assignments with operations and state transitions derived from "if" statements.

Left click on Cancel to close RTL Scheduling & Binding window.

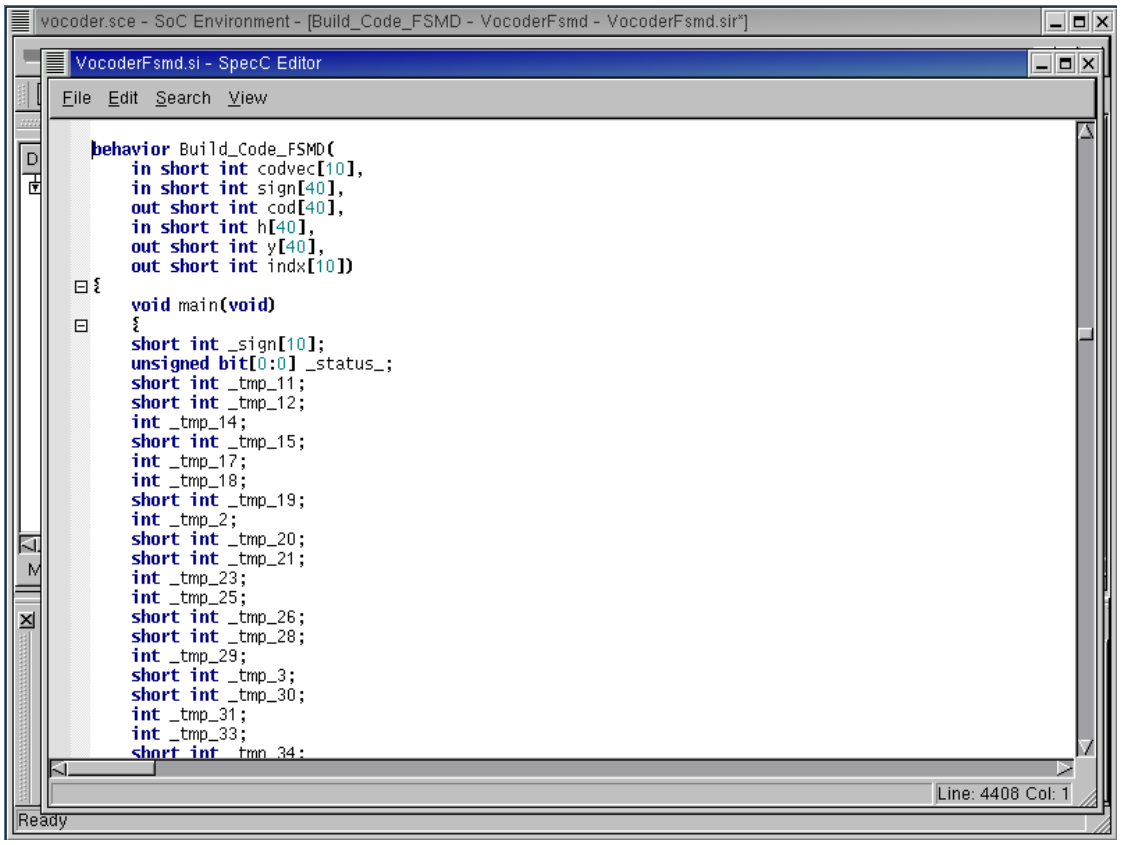
4.2.4. View SFSMD model (optional)



We browsed through the newly created model in the RTL Scheduling & Binding window. In addition, we can also view the source code of the model. Note that if reader is not interested, she or he can skip this section to go directly Section 4.2.5 *Simulate SFSMD model (optional)* (page 155).

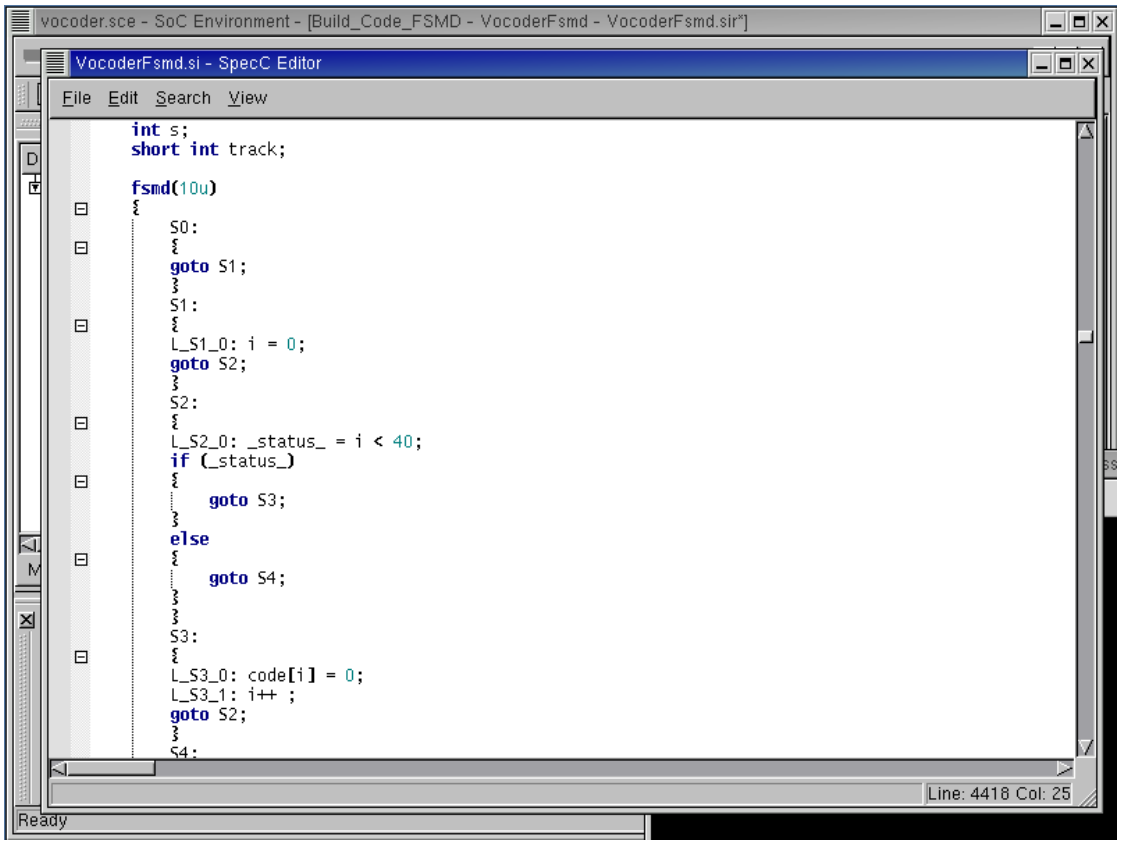
Select behavior "Build_Code_FSMD" by left clicking on it. We now take a look at the source code to see if the RTL preprocessing tool has correctly generated the SFSMD model. Do this by selecting View⟶Source from the menu bar.

4.2.4.1. View SFSMD model (optional) (cont'd)



The SpecC Editor window pops up showing the source code for behavior "Build_Code_FSMD".

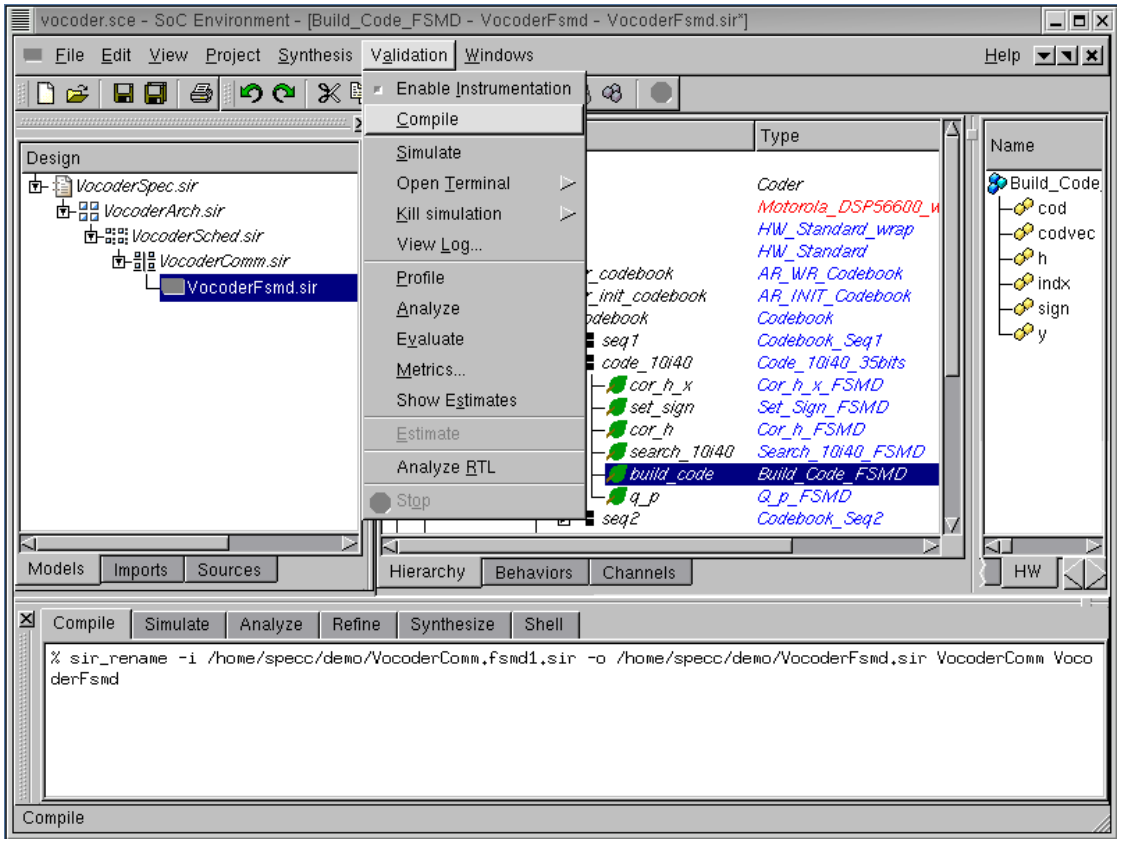
4.2.4.2. View SFSMD model (optional) (cont'd)



The behavioral input model is changed to the SFSMD model with clock period 10 ns. Scroll down the window to find loops and conditional branch constructs in the behavioral input model are changed to state transitions. Still, each state has a lot of assignments and operations, which have to be scheduled and bound.

Close the SpecC Editor window by selecting File⟶Close from the menu bar.

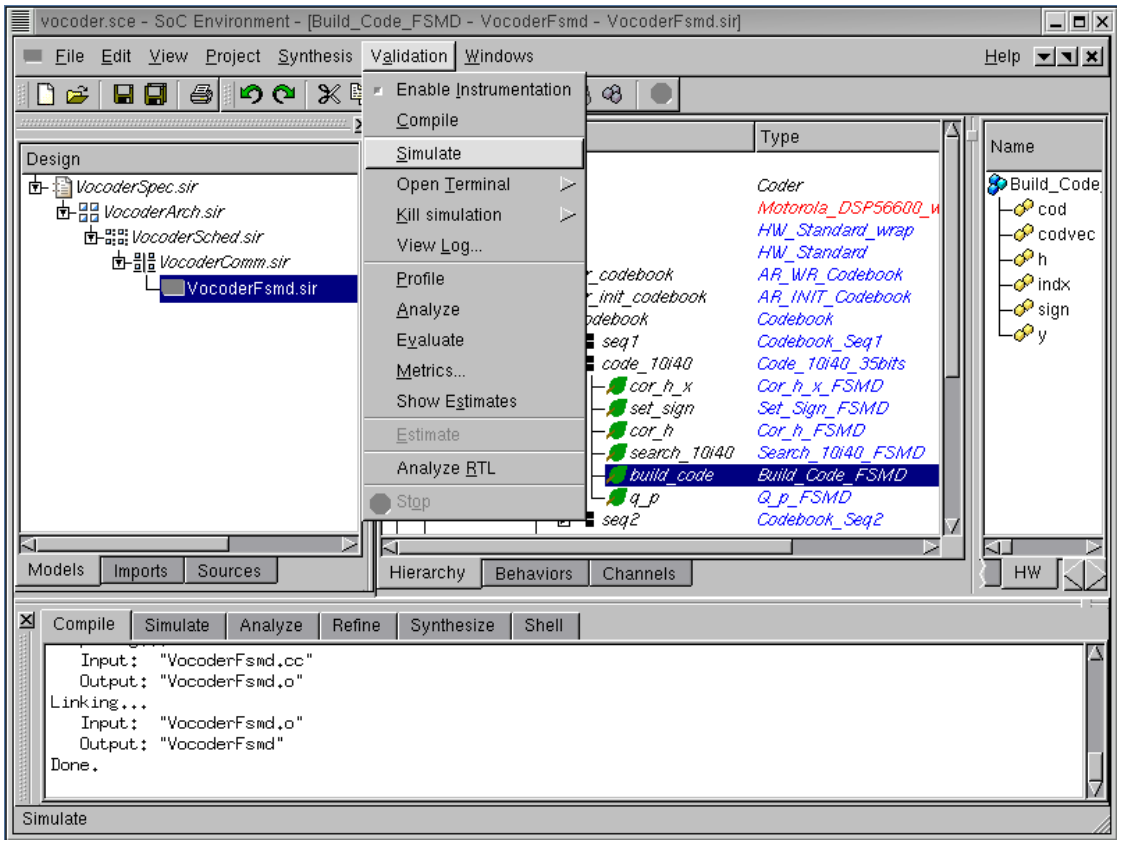
4.2.5. Simulate SFSMD model (optional)



For demo purposes, we will skip the SFSMD generation of those other behaviors assigned to HW component. Even this partially refined model is actually simulatable. To show this, first compile the model by selecting Validation⟶Compile from the menu bar.

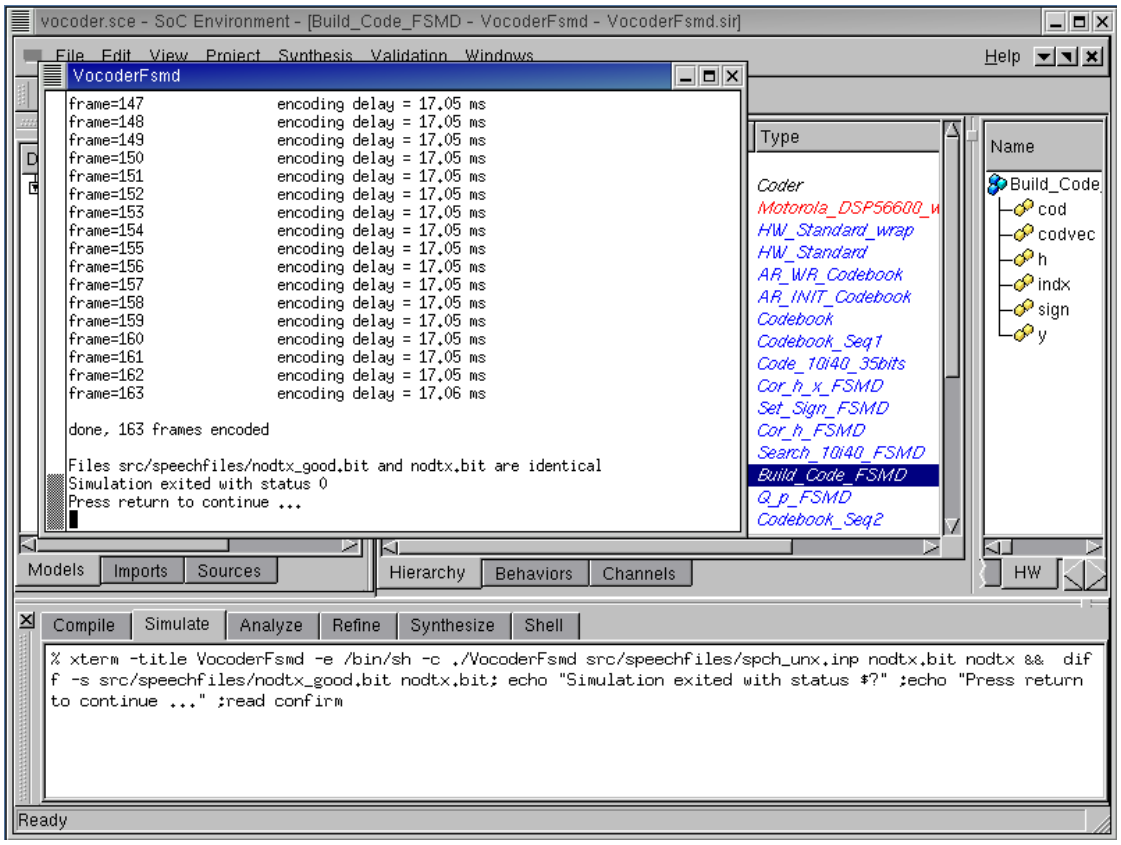
If reader is not interested, she or he can skip this section to go directly Section 4.2.6 *Analyze SFSMD model* (page 158).

4.2.5.1. Simulate SFSMD model (optional) (cont'd)



Note that the SFSMD model compiles correctly into executable "VocoderFSMD" as seen in the logging window. We now proceed to simulate the model by selecting Validation⟶Simulate from the menu bar.

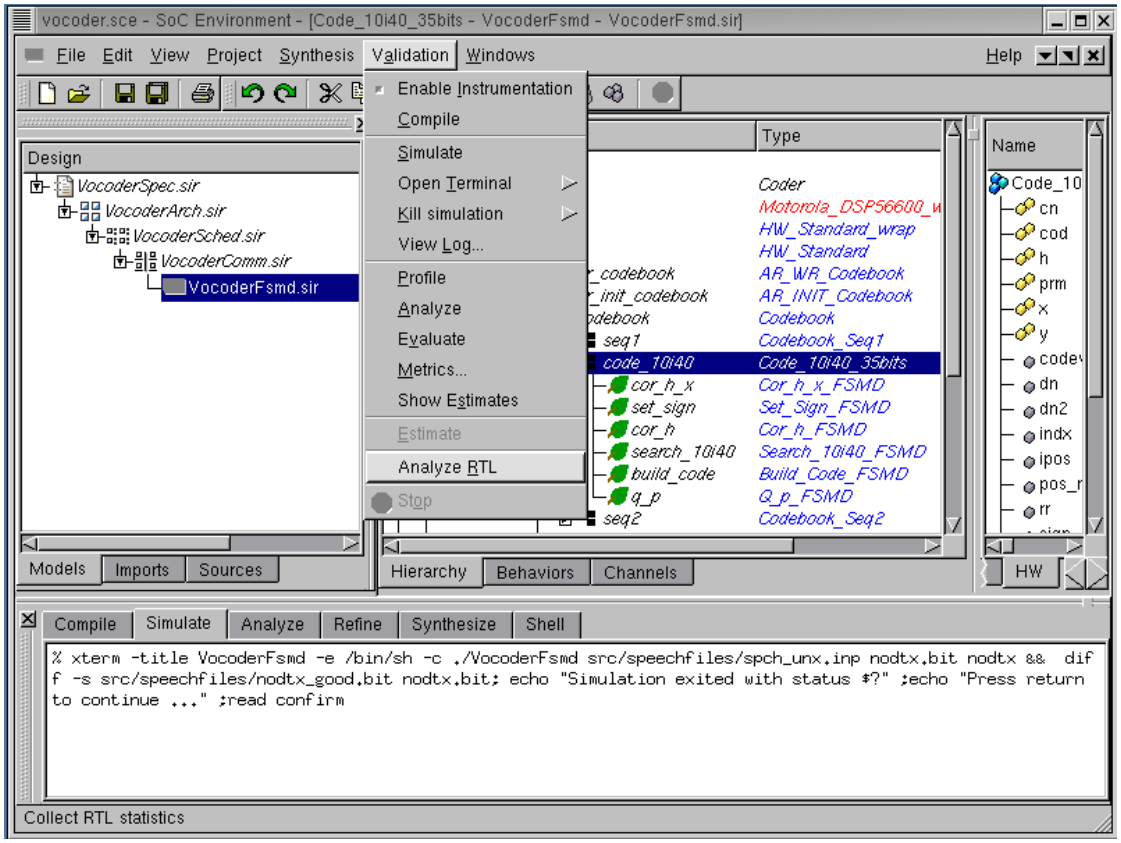
4.2.5.2. Simulate SFSMD model (optional) (cont'd)



The simulation window pops up showing the progress and successful completion of simulation. We are thus ensured that the SFSMD generation step has taken place correctly. Also note that we can perform the SFSMD generation on any behavior of our choice. This indicates that the user has complete freedom of delving into one behavior at a time and testing it thoroughly. Since the other behaviors are at higher level of abstraction, the simulation speed is much faster than the situation when the entire model is synthesized. This is a big advantage with our methodology and it enables partial simulation of the design. The designer does not have to refine the entire design to simulate just one behavior in RTL.

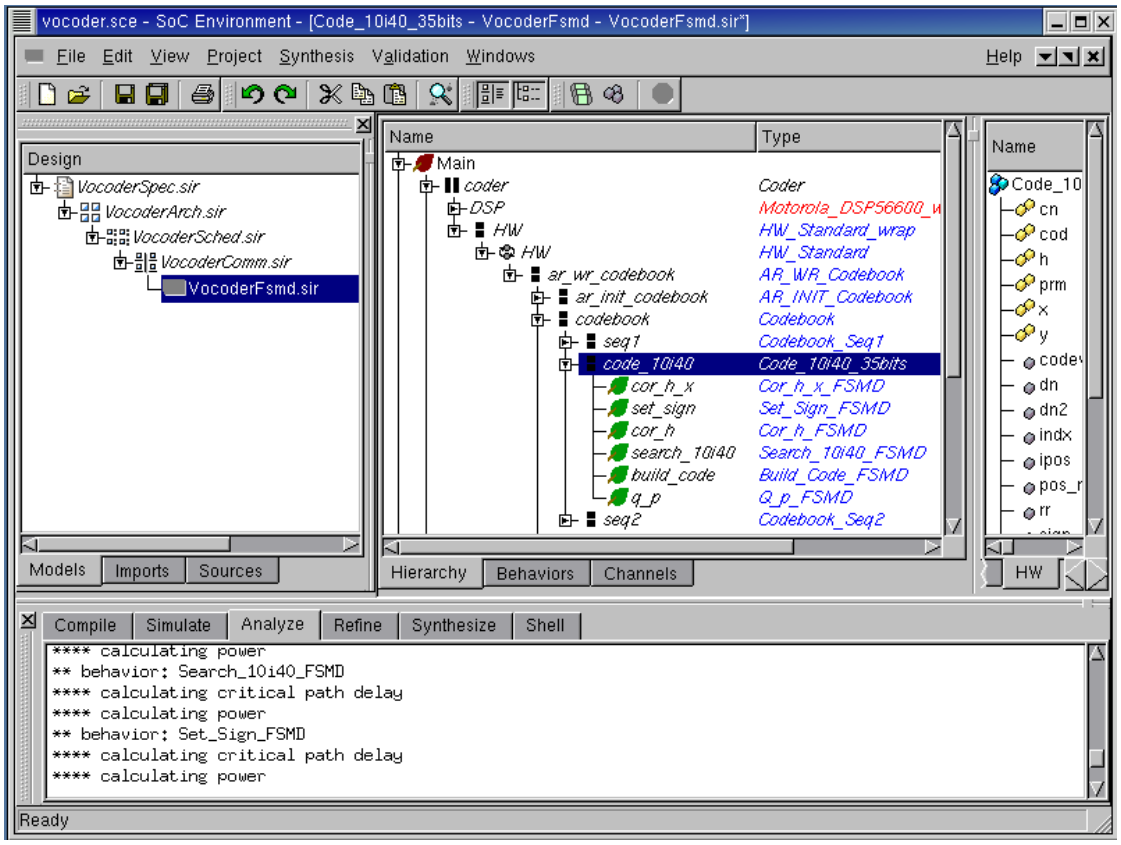
In this simulation, we see the delay per frame in the SFSMD model decreases to 17.05 ns from 19.89 ns compared to the communication model. Because each state in the SFSMD model is artificially assigned a 10 ns clock period even though it has a lot of assignments and operations to be split into multiple states by scheduling and binding.

4.2.6. Analyze SFSMD model



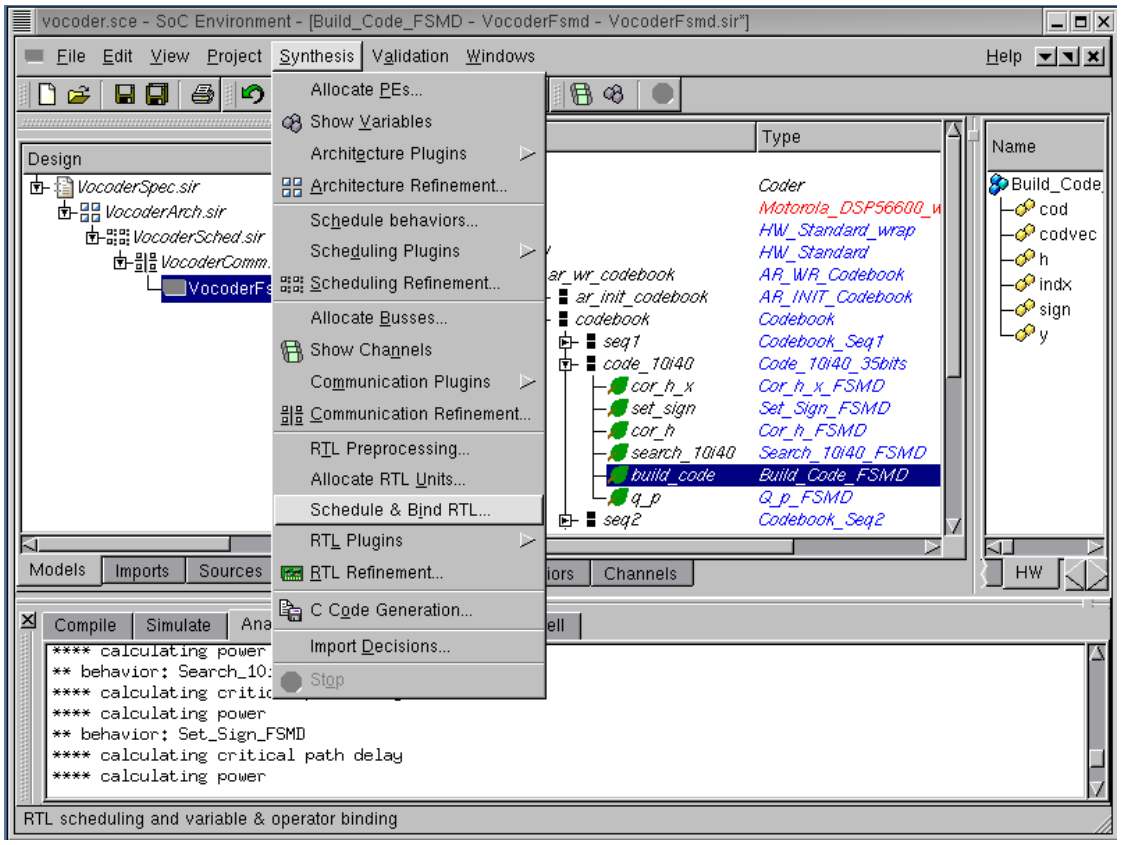
Once the SFSMD model is generated, we need to allocate RTL components. For allocation, we need to get some statistical information on design. The statistical information contains the number of operations for functional unit allocation, the number of live variables for storage unit allocation and the number of data transfers for bus allocation and the number of operations in critical path in each state. These kind of useful information can be obtained by performing RTL analysis. First we select the behavior "Code_10i40_35bits", of which we want to get the statistical information. The RTL analysis is performed by selecting Validation⟶Analyze RTL from the menu bar.

4.2.6.1. Analyze SFSMD model (cont'd)



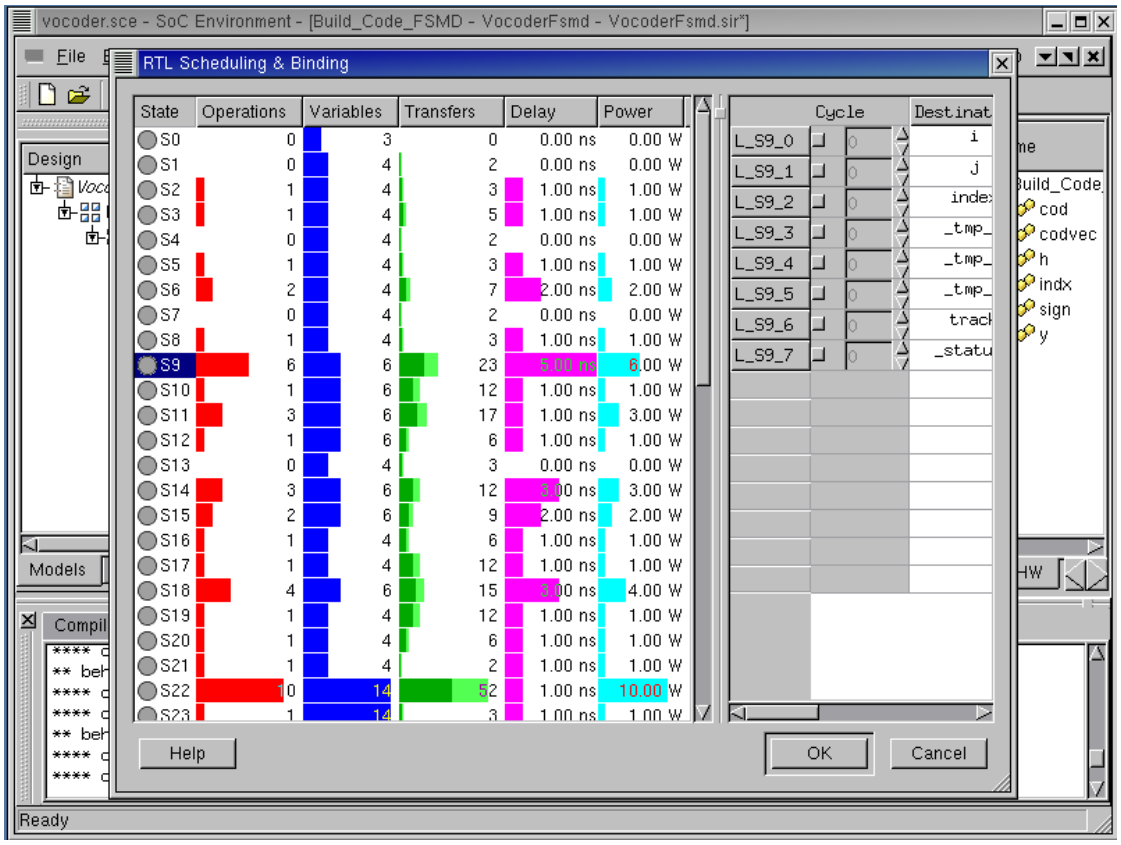
RTL analysis tool goes over all sub-behaviors in the behavior "Code_10i40_35bits", and generates their statistical information for the allocation.

4.2.6.2. Analyze SFSMD model (cont'd)



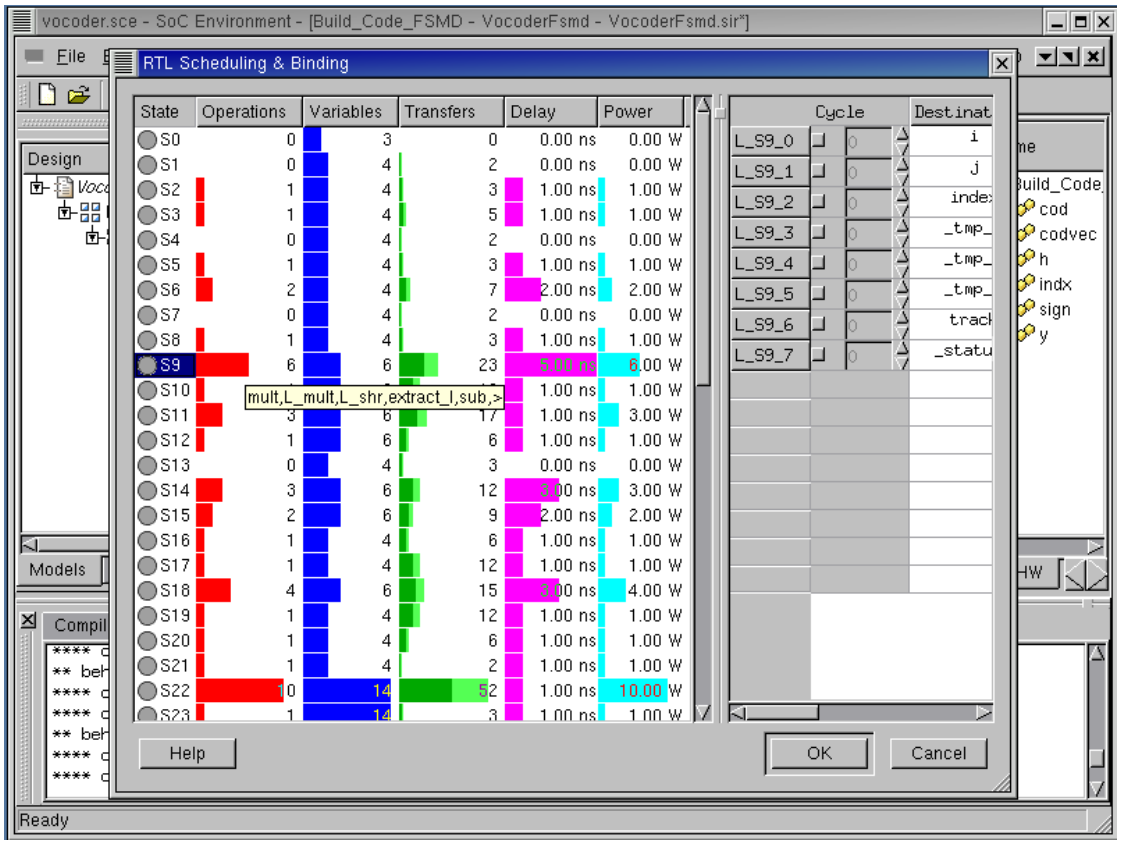
In order to look at RTL analysis result for the behavior "Build_Code_FSMD", select Synthesis⟶Schedule & Bind RTL from the menu bar.

4.2.6.3. Analyze SFSMD model (cont'd)



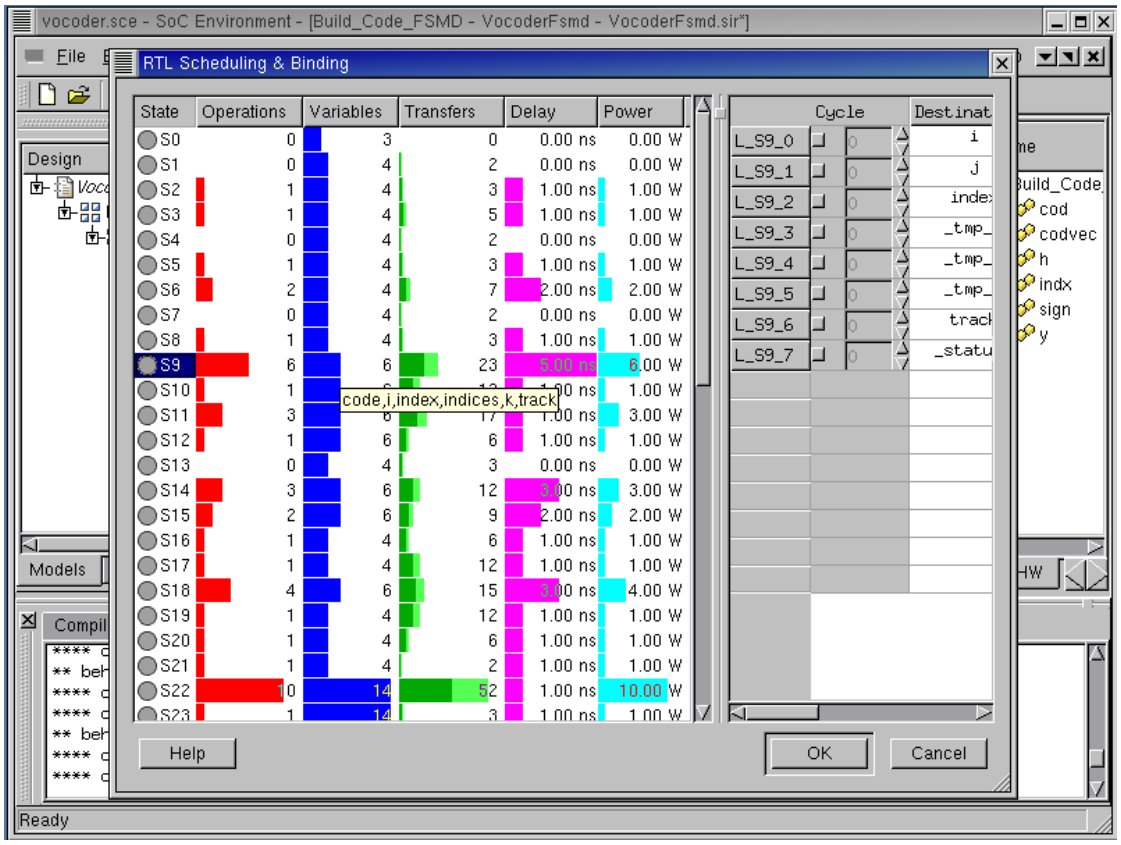
The RTL Scheduling & Binding window pops up showing the statistical information for the selected behavior. From left to right in the left panel of the RTL Scheduling & Binding window, it shows number of operations (Operations column) in each state, number of variables (Variables), number of data transfers (Transfers), number of operations in critical path (Delay), and power dissipation (Power).

4.2.6.4. Analyze SFSMD model (cont'd)



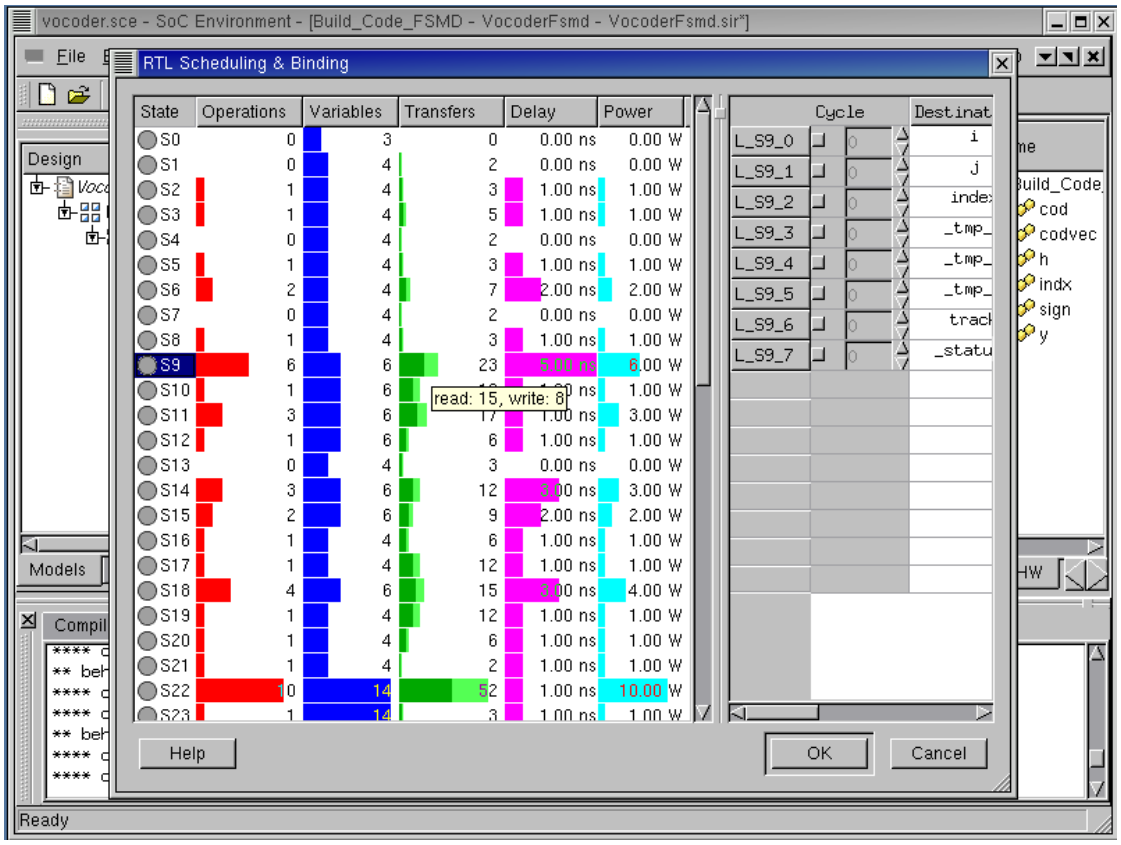
Moving the mouse over the bars in the graph gives us detailed information on each category. For instance, if we put the mouse over the Operations column in each state, the operations which are executed in the state will be shown like mult, L_mult, L_shr, extract_l, sub and > in state S9.

4.2.6.5. Analyze SFSMD model (cont'd)



If we move the mouse over the Variables column in each state, the variables which are live at the end of the state will be shown like code, i, index, indices, k, and track in state S9.

4.2.6.6. Analyze SFSMD model (cont'd)



If we move the mouse over the Transfers column in each state, the data transfers happens at the state will be shown. In state S9, the number of read transfers is 15 and the number of write transfers, 8.

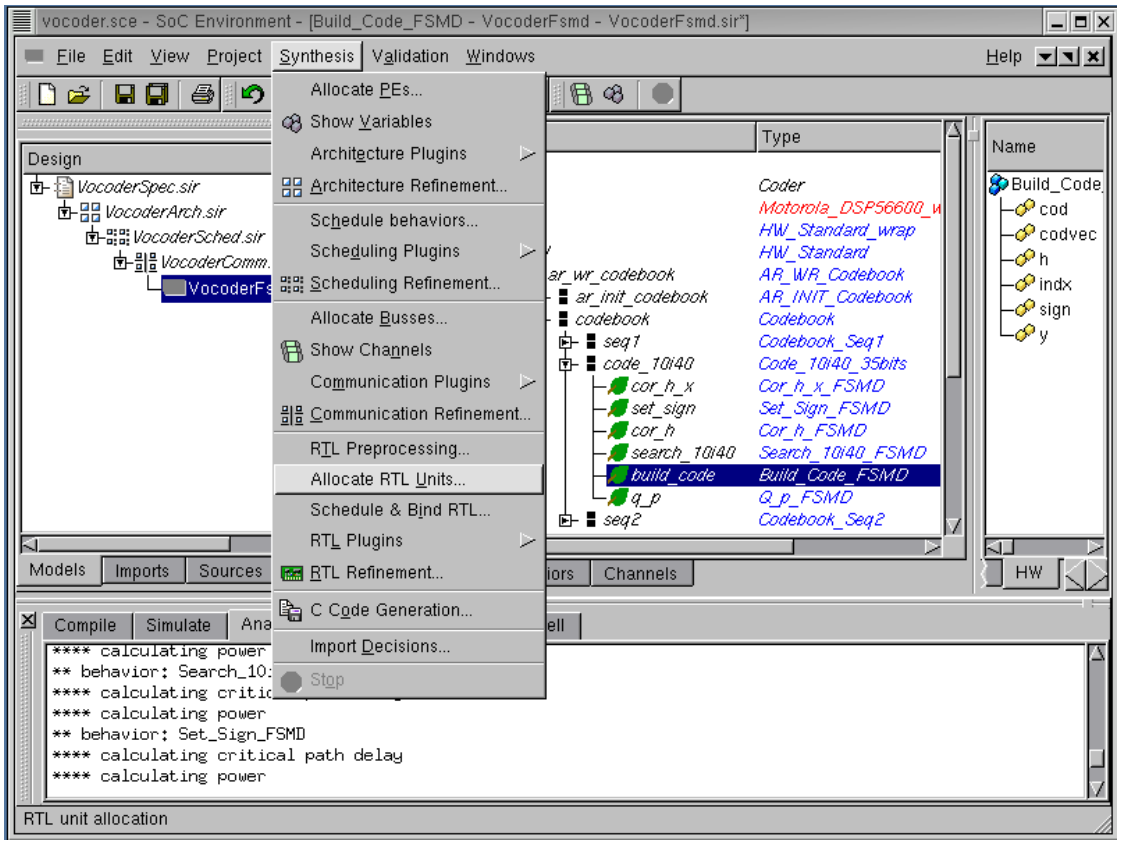
Left click on Cancel to close the RTL Scheduling & Binding.

4.3. RTL Allocation

RTL allocation is one of important steps for custom hardware design. It is to select number of RTL components for the design, while meeting various constraints. For RTL allocation, we need to get a statistical information on the design.

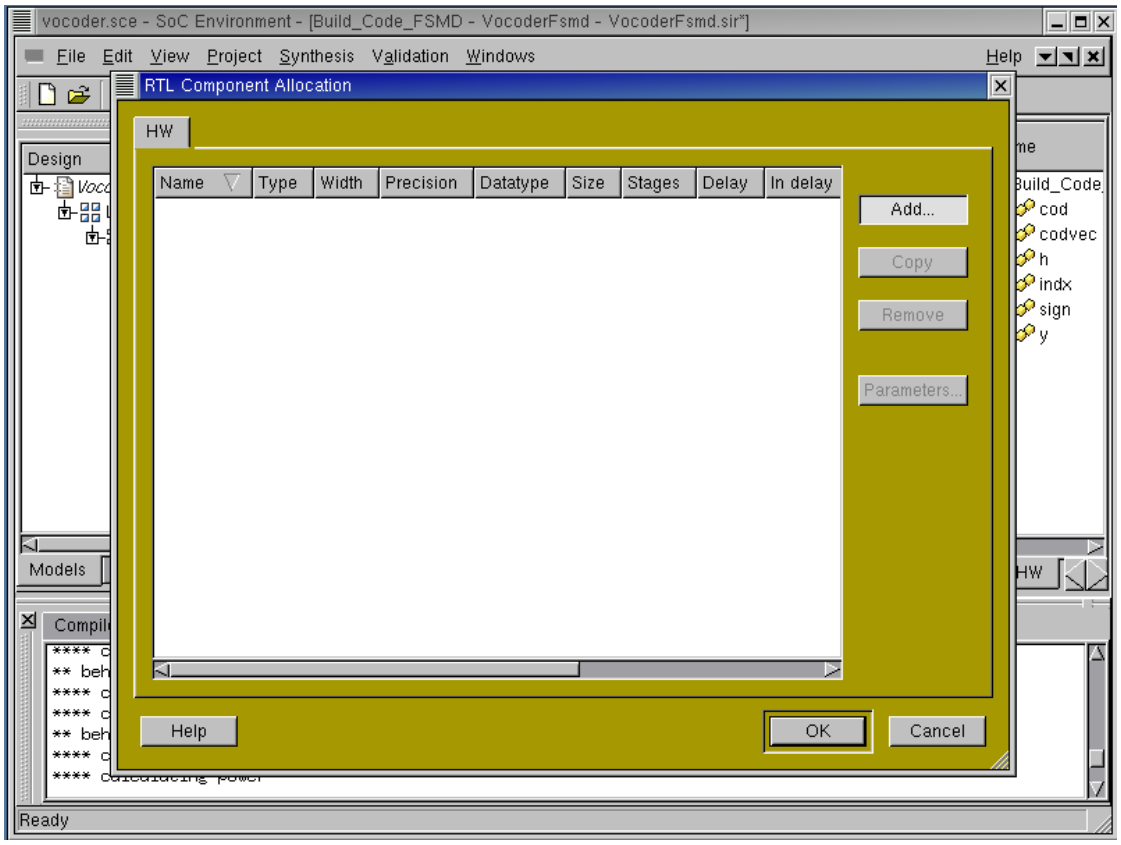
The statistical information contains the number of operations for functional unit allocation, the number of live variables for storage unit allocation and the number of data transfers for bus allocation and the number of operations in the critical path in each state. These kinds of information can be obtained by performing RTL analysis.

4.3.1. Allocate functional units



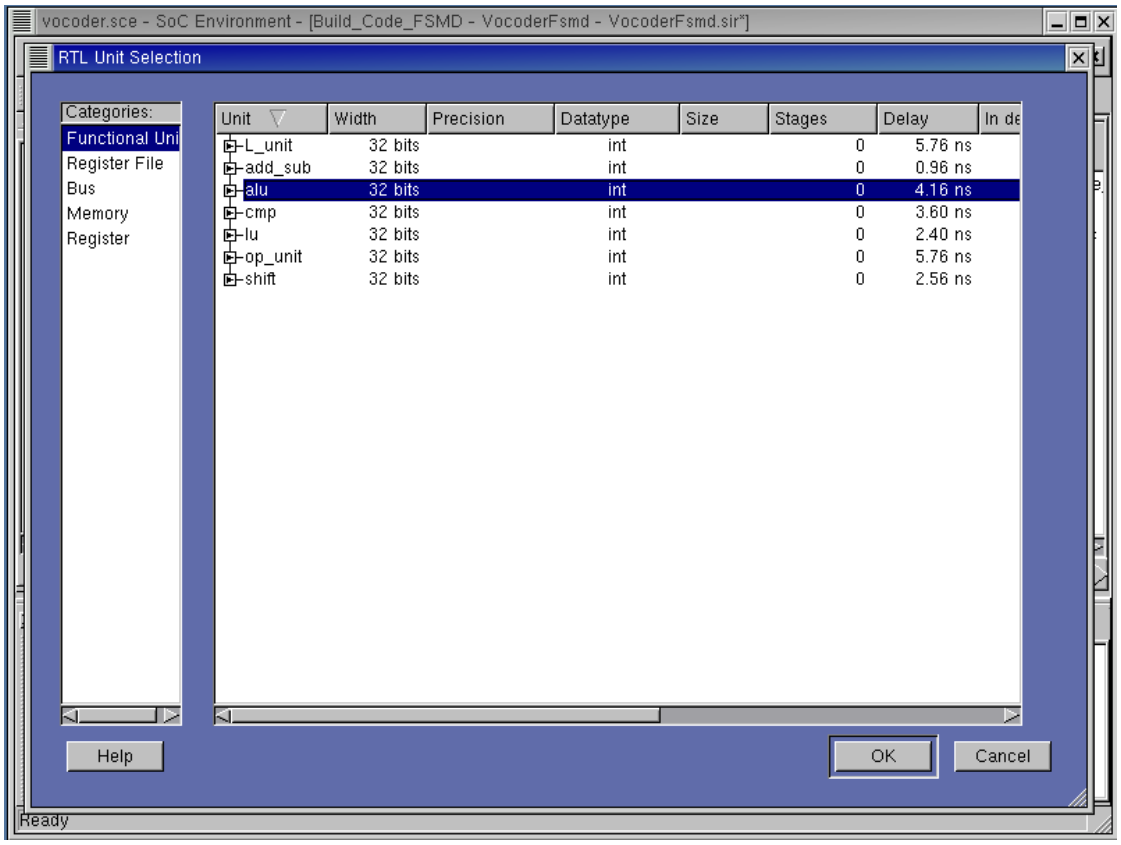
After we produce a valid SFSMD model during preprocessing step, the next step is to allocate RTL components for HW part of the system. The allocation will be guided by RTL statistical information. To perform the allocation, select Synthesis⟶Allocate RTL Units from the menu bar.

4.3.1.1. Allocate functional units (cont'd)



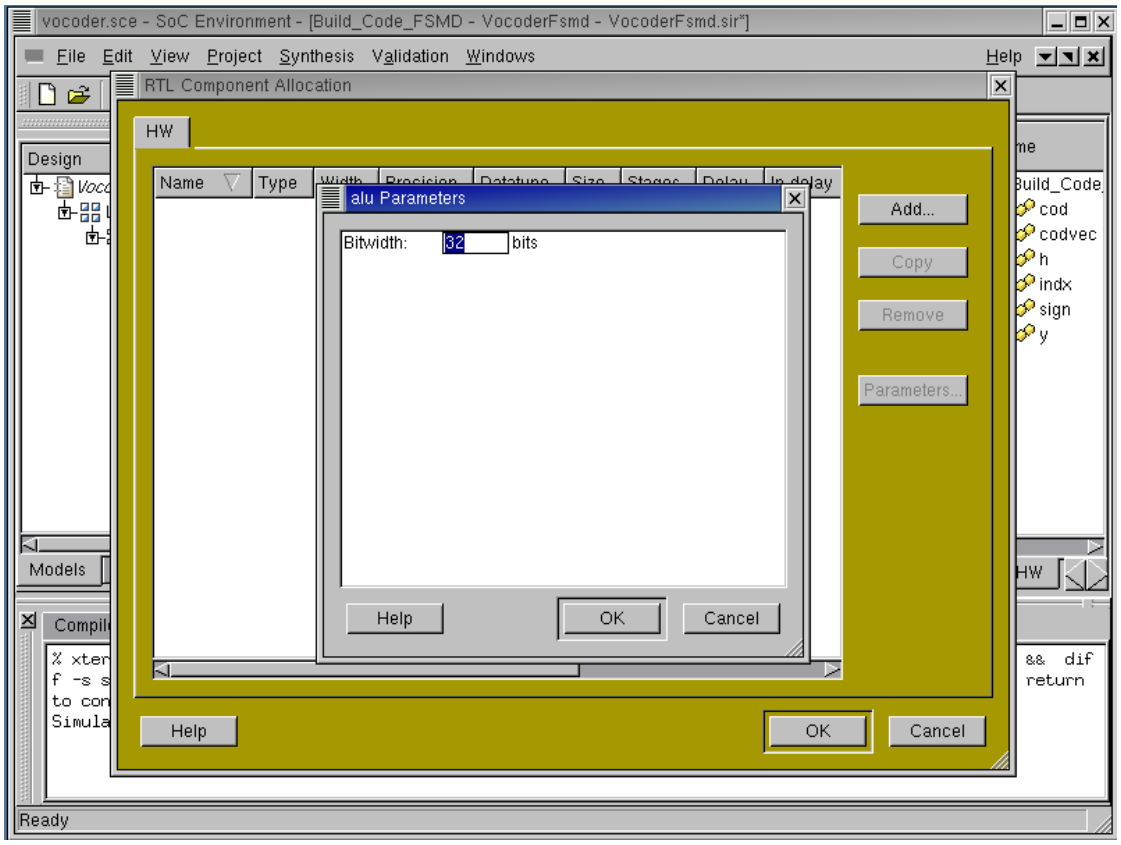
An RTL allocation window pops up just like for components and busses. left click on Add to see the include units from the database into the design.

4.3.1.2. Allocate functional units (cont'd)



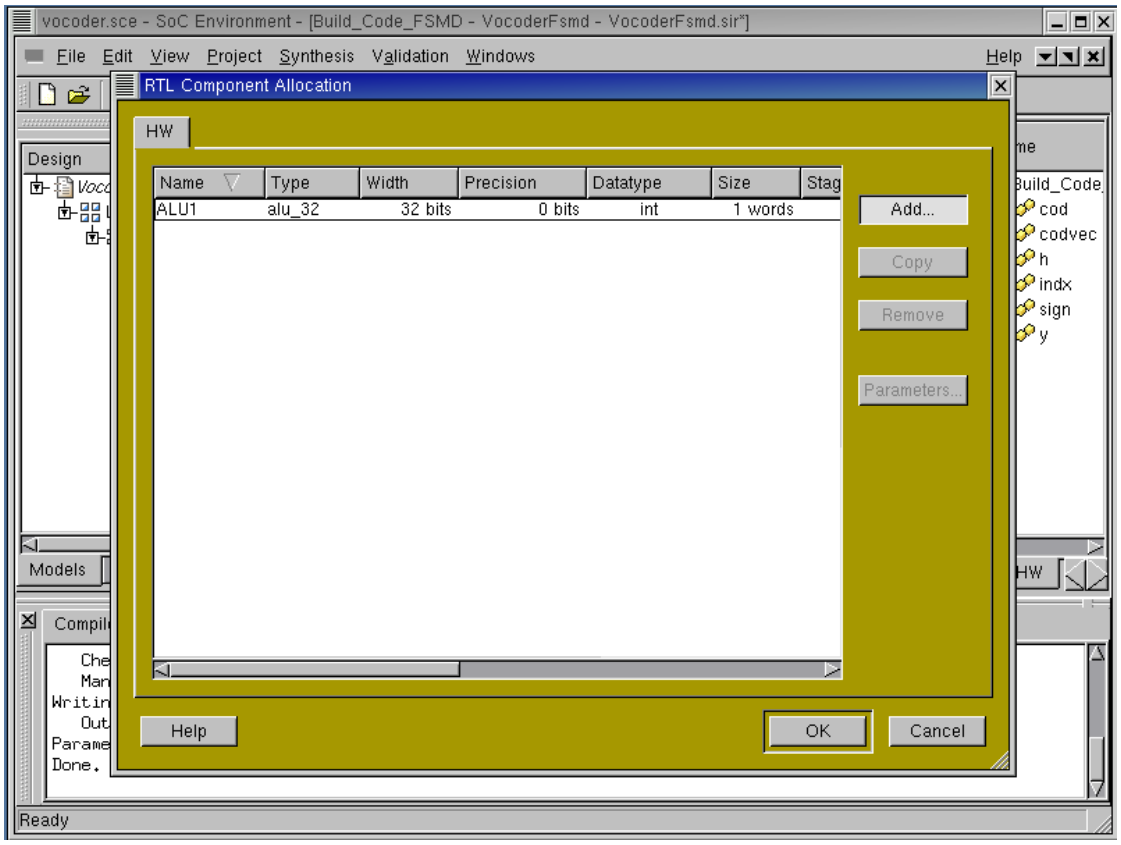
A RTL Unit Selection window pops up for RTL unit selection. There are various categories for the RTL components listed on the left-most column. Left click on "Functional Unit" to see the functional units and their parameters in the right-most column. In this tutorial, we will select 3 functional units: "L_unit" and "op_unit" for saturated arithmetic operations and "alu" for the other operations. To select an alu, left click on "alu" and click on OK to add it to RTL Unit Selection window.

4.3.1.3. Allocate functional units (cont'd)



A new property box for the alu component pops up and shows the configurable parameters. In case of alu, bit width is the configurable parameter. Left click on OK to use the default value of 32 bits.

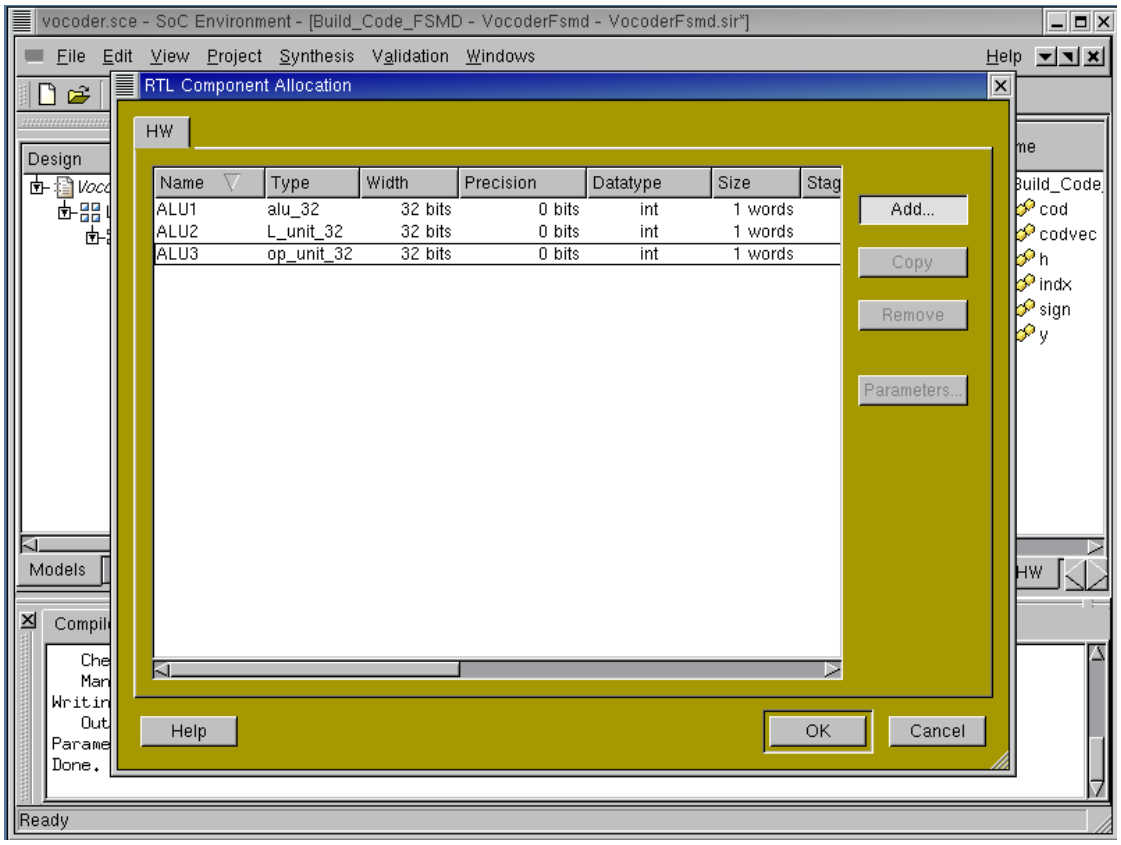
4.3.1.4. Allocate functional units (cont'd)



The allocated alu component will be shown in the RTL Component Allocation window. Left click on Name column of the allocated alu to rename it to ALU1.

We may repeat the last procedure to allocate more RTL components from the database.

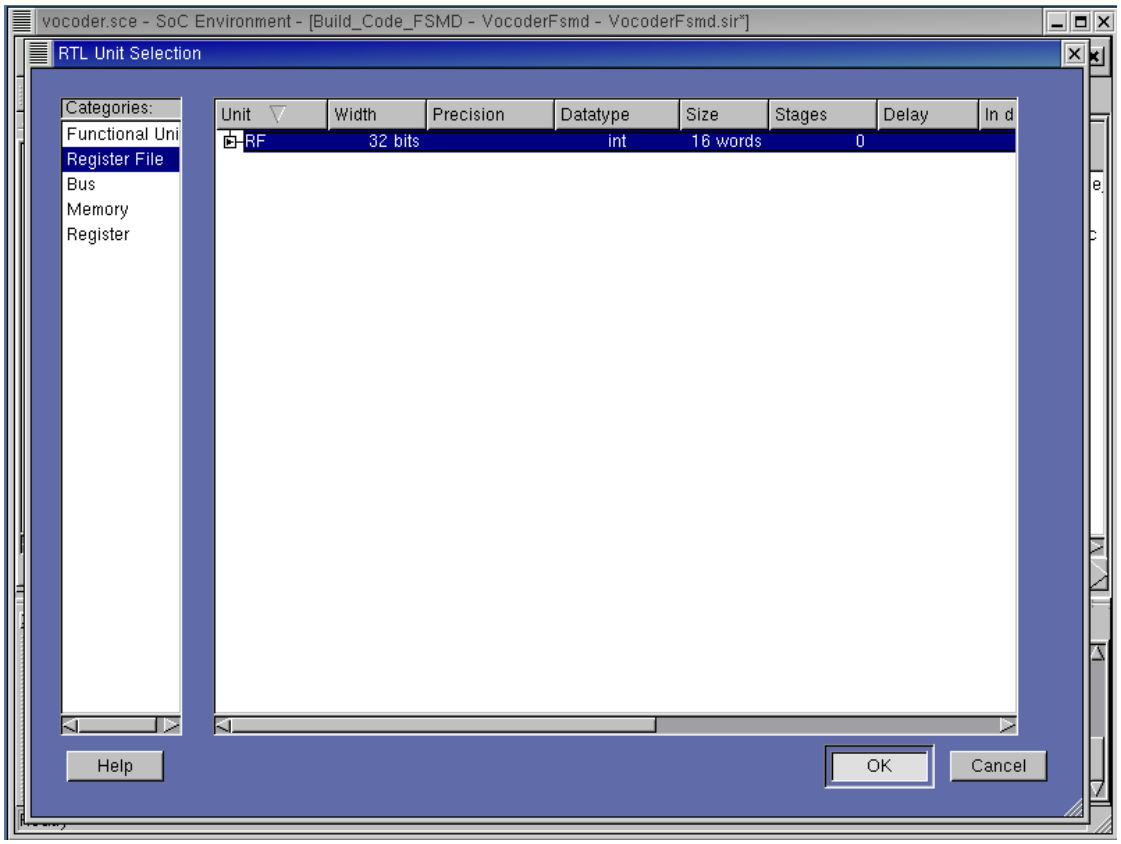
4.3.1.5. Allocate functional units (cont'd)



In this way, we can allocate an "L_unit" and an "op_unit" and rename them to ALU2 and ALU3 respectively.

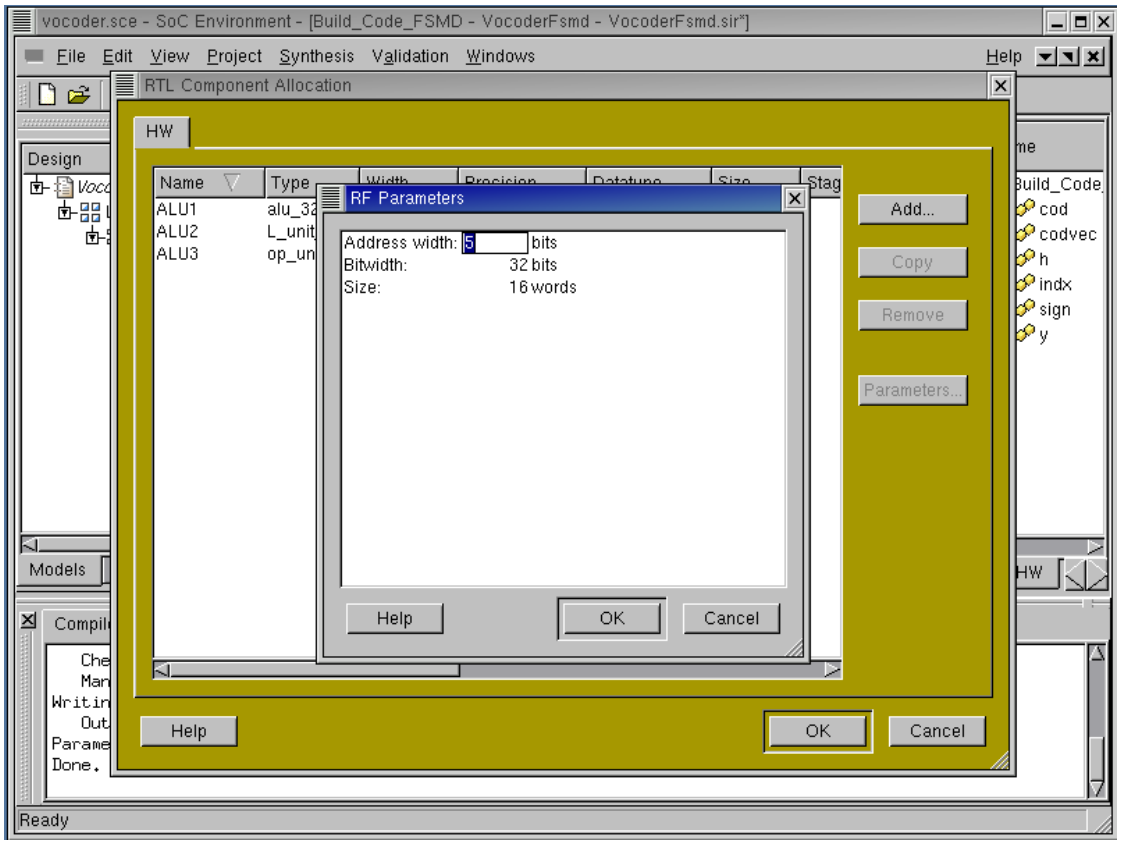
All desirable functional units for hardware implementation have now been selected. However, we also need storage units like register files and memory. Left click on Add.

4.3.2. Allocate storage units



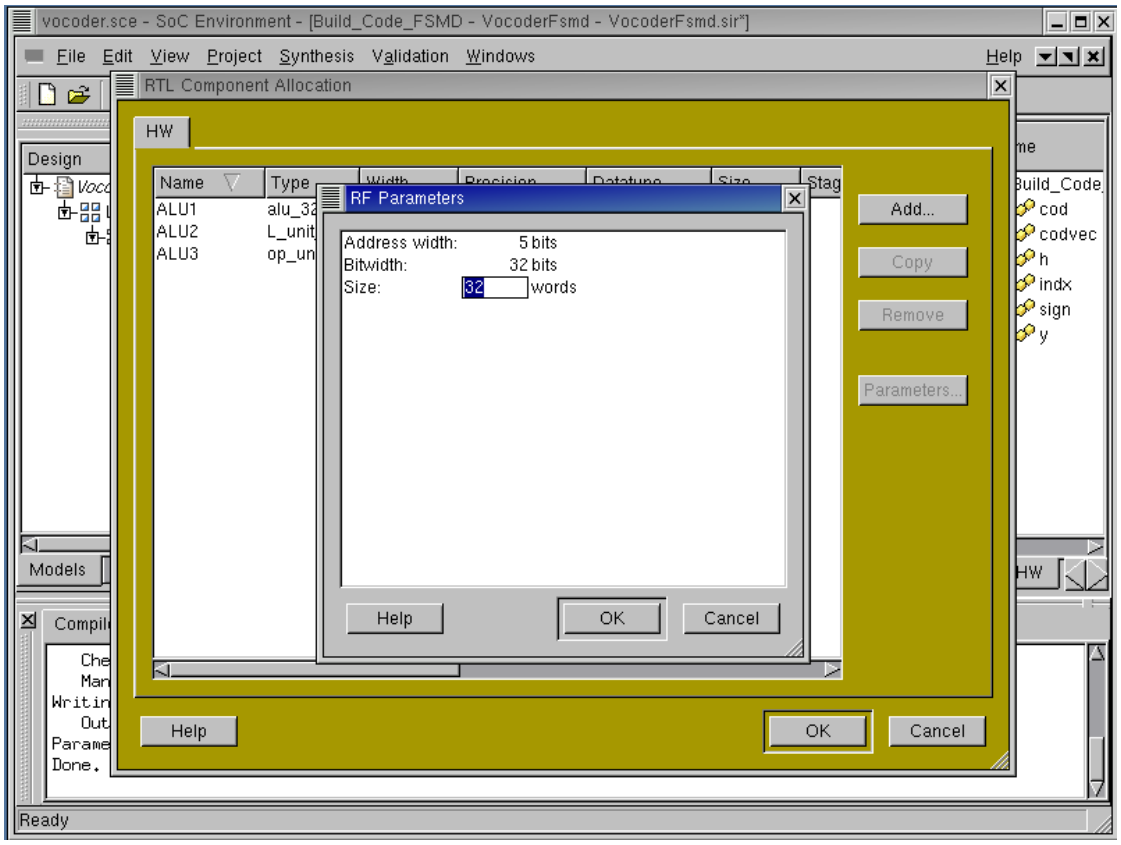
Left click on "Register File" to see the various register files and their properties. Left click on "RF" to select register file and click on OK to add it to RTL Unit Selection window.

4.3.2.1. Allocate storage units (cont'd)



A new property box for RF component pops up and shows the configurable parameters. In case of RF, address width and size of register file as well as bit width are the configurable parameters. Left click on "Address width" to change 4 bits to 5 bits.

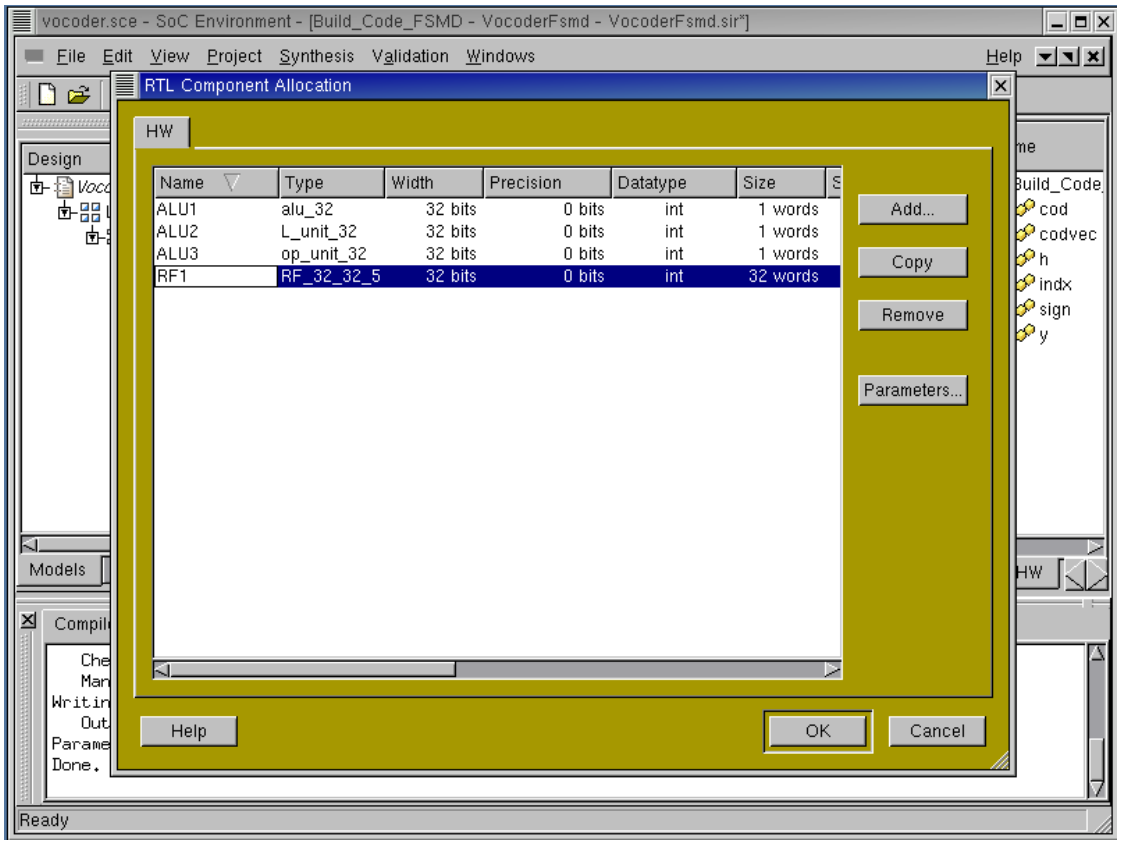
4.3.2.2. Allocate storage units (cont'd)



Since the address width is changed to 5 bits, the allowed address space is 32 words. Left click on size to change 16 words to 32 words.

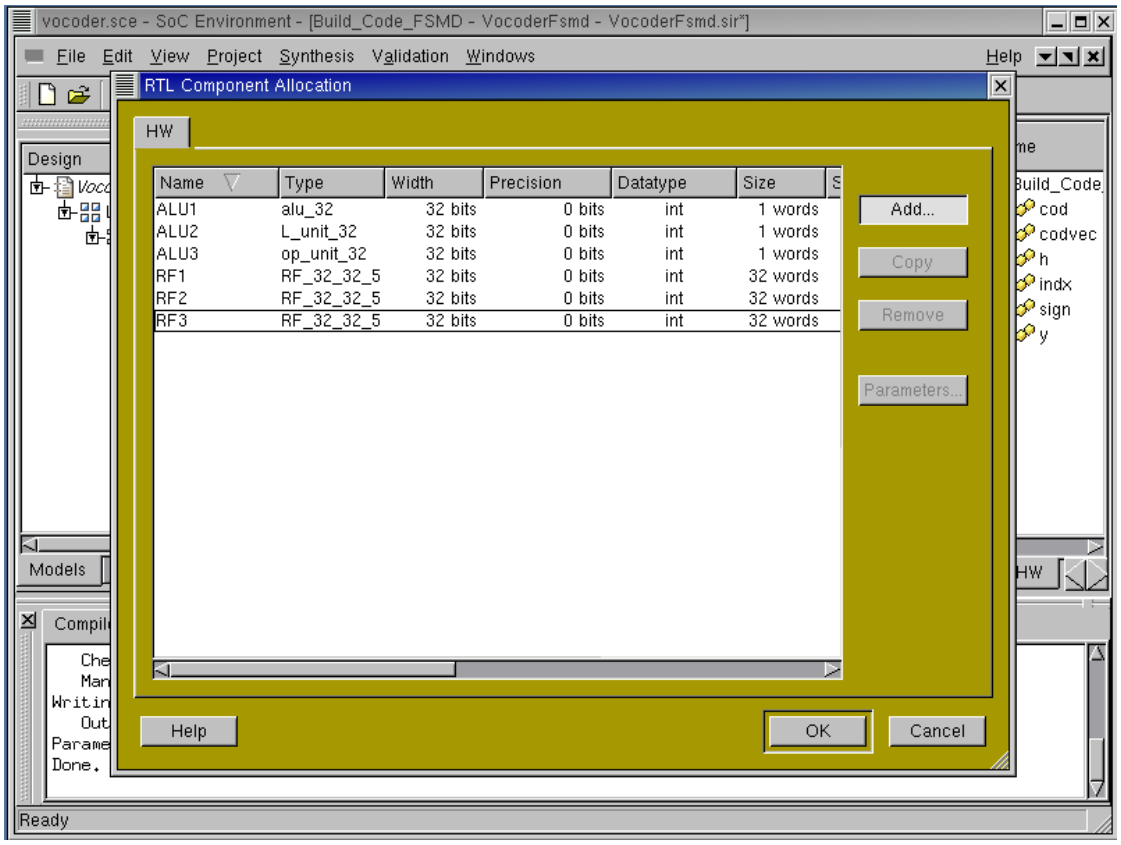
Left click on OK to add RF to RTL allocation.

4.3.2.3. Allocate storage units (cont'd)



The selected RF component will be shown in the RTL Component Allocation window. Left click on Name column of the allocated RF to rename it to RF1.

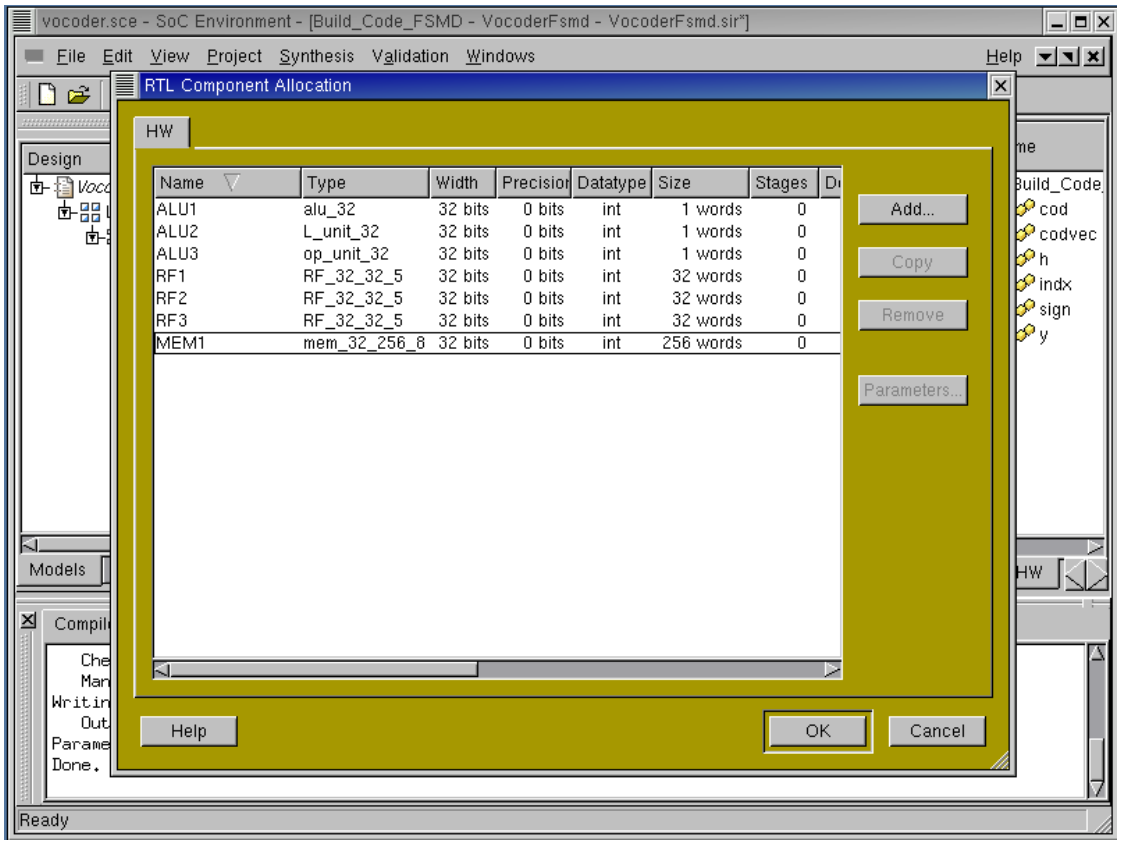
4.3.2.4. Allocate storage units (cont'd)



For the purpose of this design we will need 3 register files to perform RTL synthesis. To add more register files in the allocation table, simply Left click on Copy by 2 times. This is a useful way to replicate components for large sized allocations.

Now, we have allocated 3 register files. In the similar way, we can allocate a memory component.

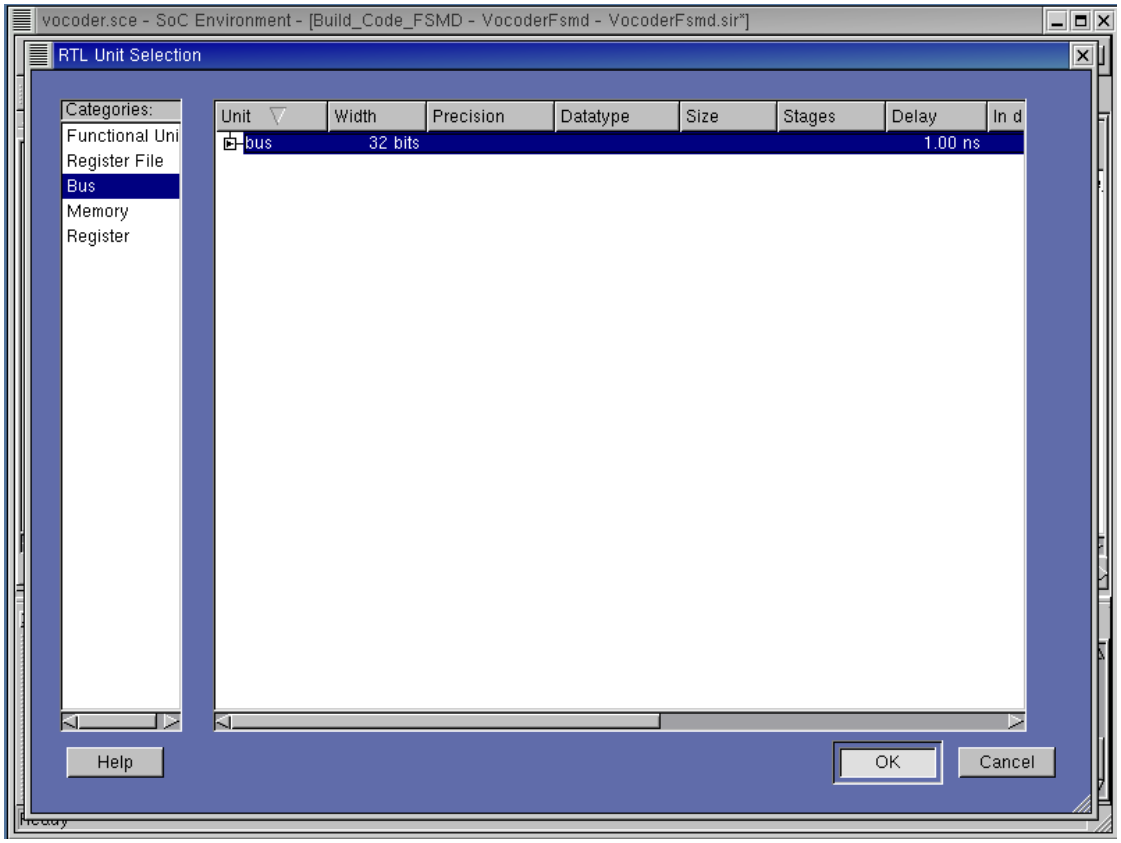
4.3.2.5. Allocate storage units (cont'd)



In the "Memory" category, we select the "mem" type memory. Its size is 256 words, and then its address width is 8 bits. Also its bit width is 32 bits.

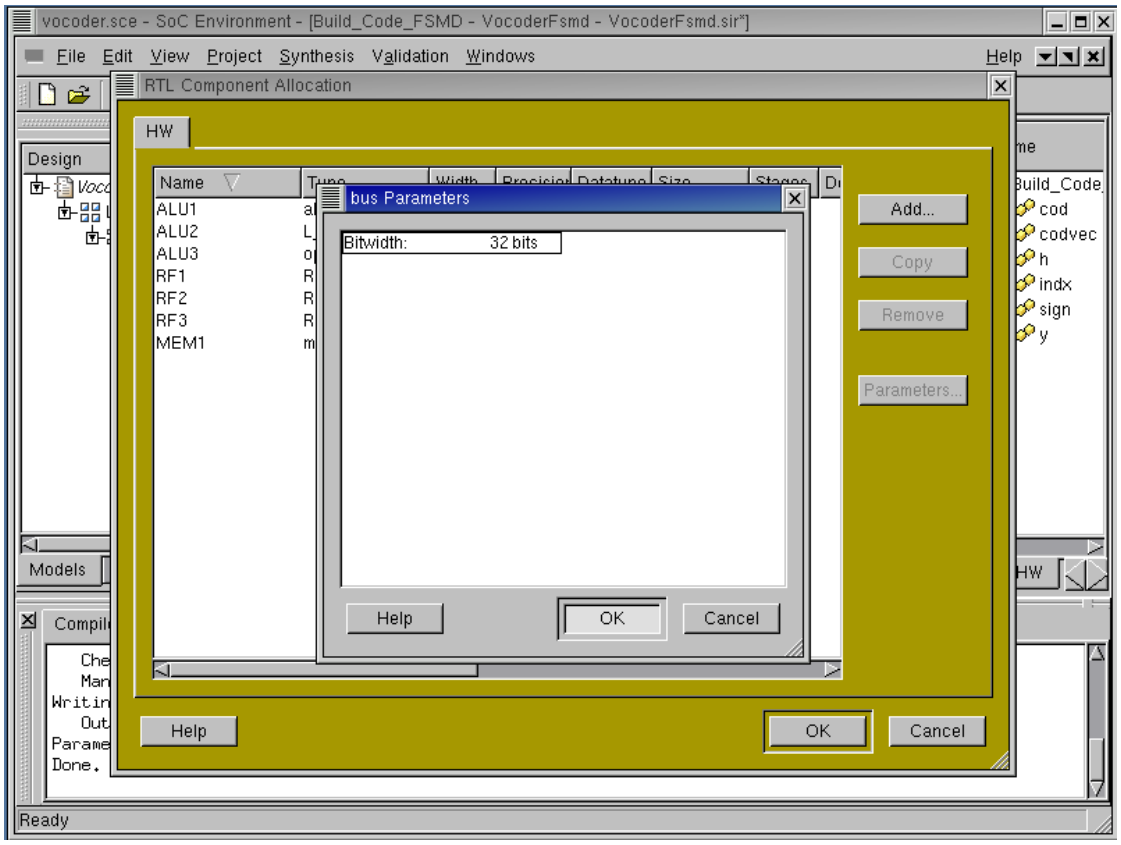
We are now done with storage unit allocation and we have to allocate busses for data transfers between storage units and functional units. Left click on Add to add more RTL components.

4.3.3. Allocate buses



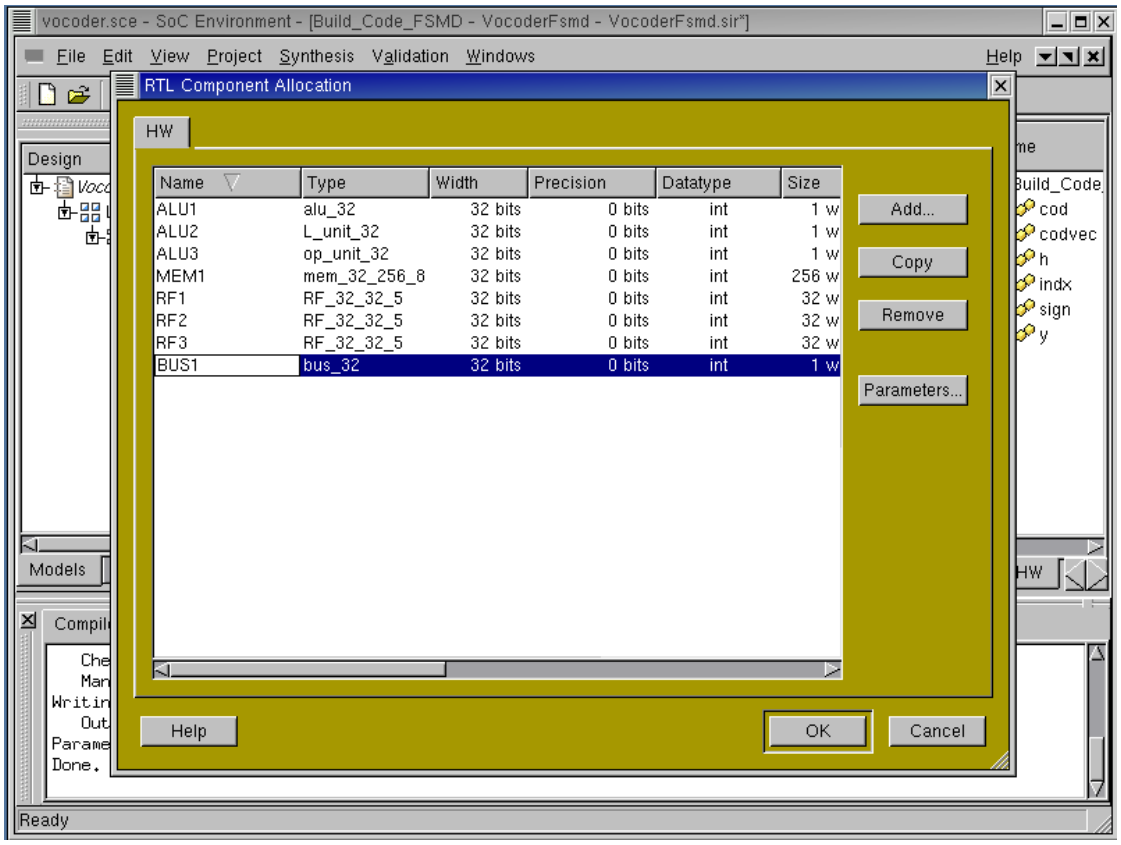
Left click on "Bus" to see its properties in the left-most column. Left click on "bus" to select the bus and press OK.

4.3.3.1. Allocate buses (cont'd)



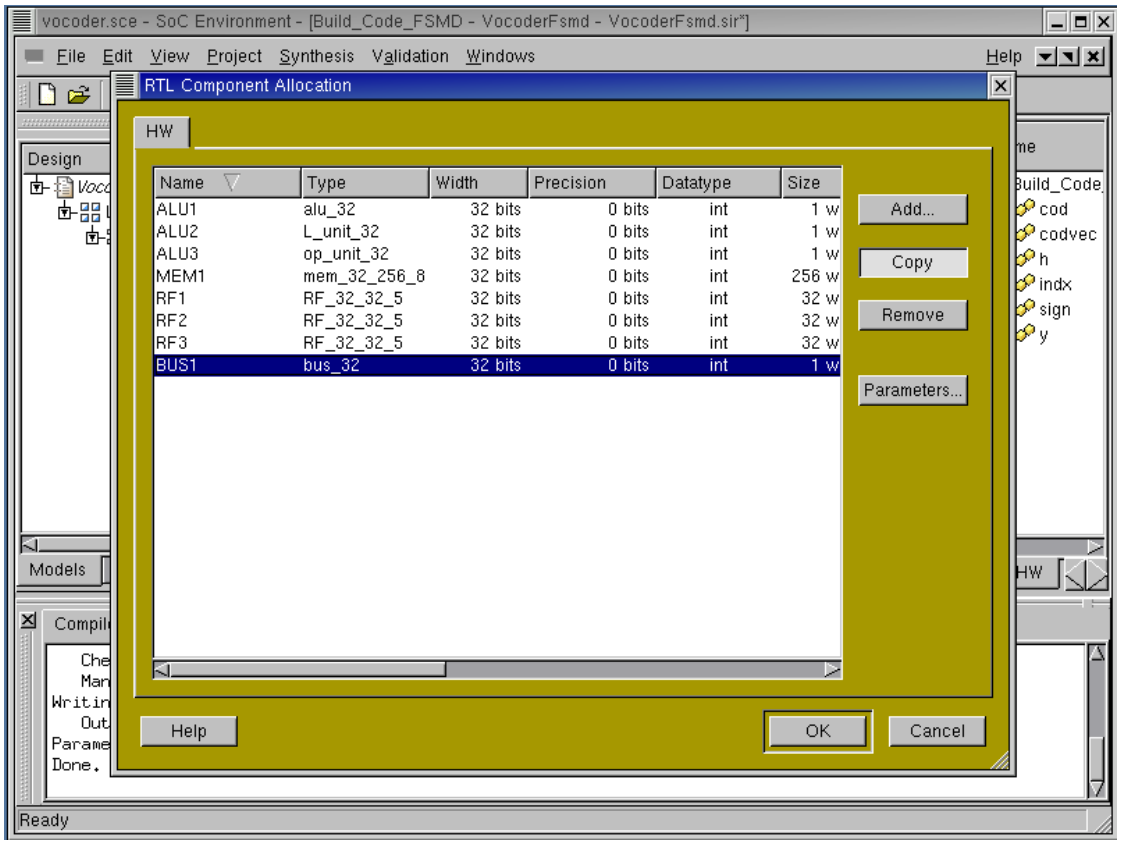
A new property box for bus component pops up and shows the configurable parameters. In case of bus, bit width is the configurable parameter. Left click on OK to add bus to RTL allocation.

4.3.3.2. Allocate buses (cont'd)



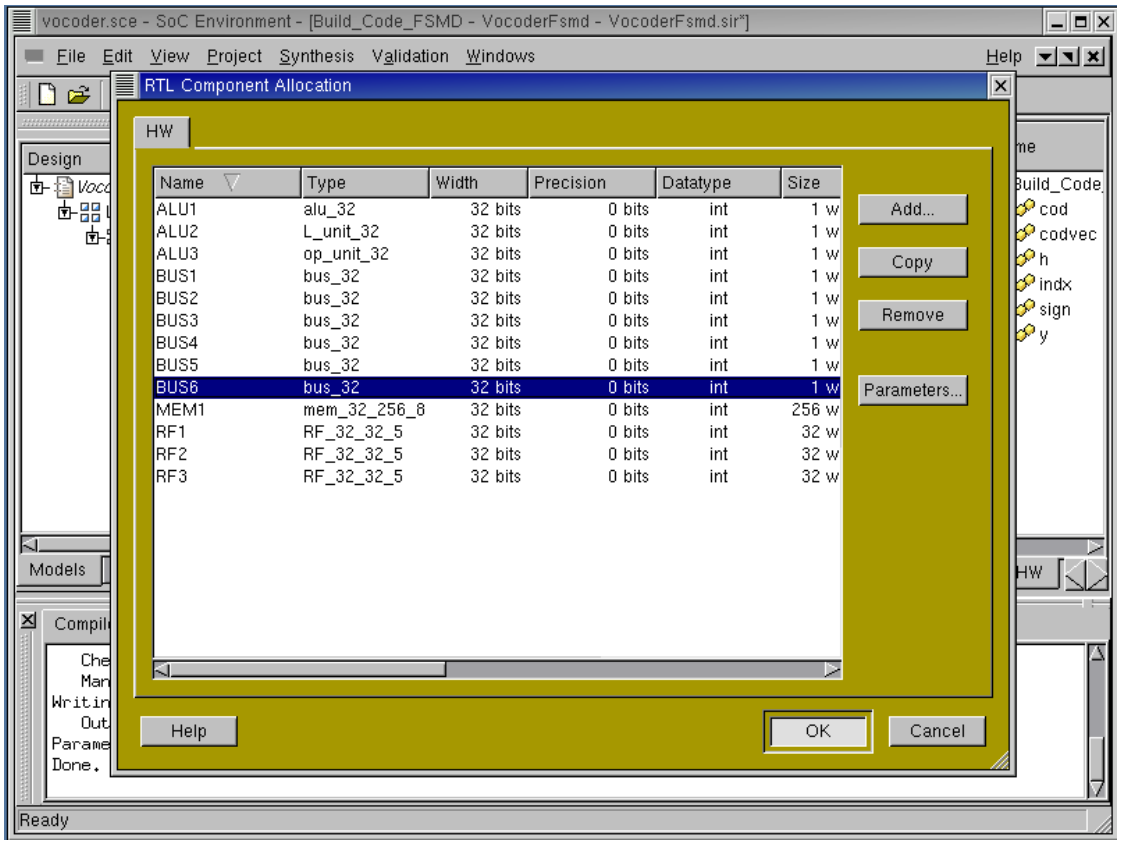
The selected bus component will be shown in the RTL Component Allocation window. Left click on Name column of the allocated bus to rename it to BUS1.

4.3.3.3. Allocate buses (cont'd)



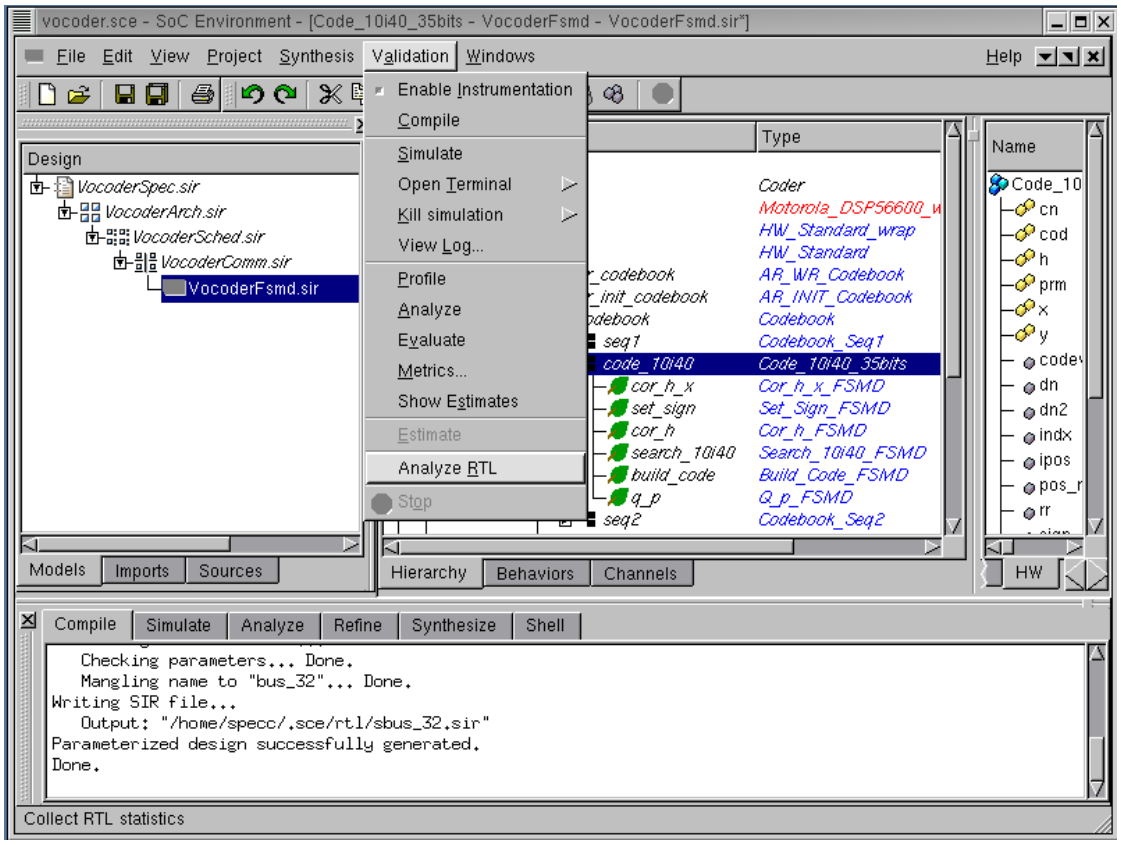
For the purpose of this design we will need 6 buses to perform RTL synthesis. To add more buses in the allocation table, simply left click on Copy by 5 times. This is a useful way to replicate components for large sized allocations.

4.3.3.4. Allocate buses (cont'd)



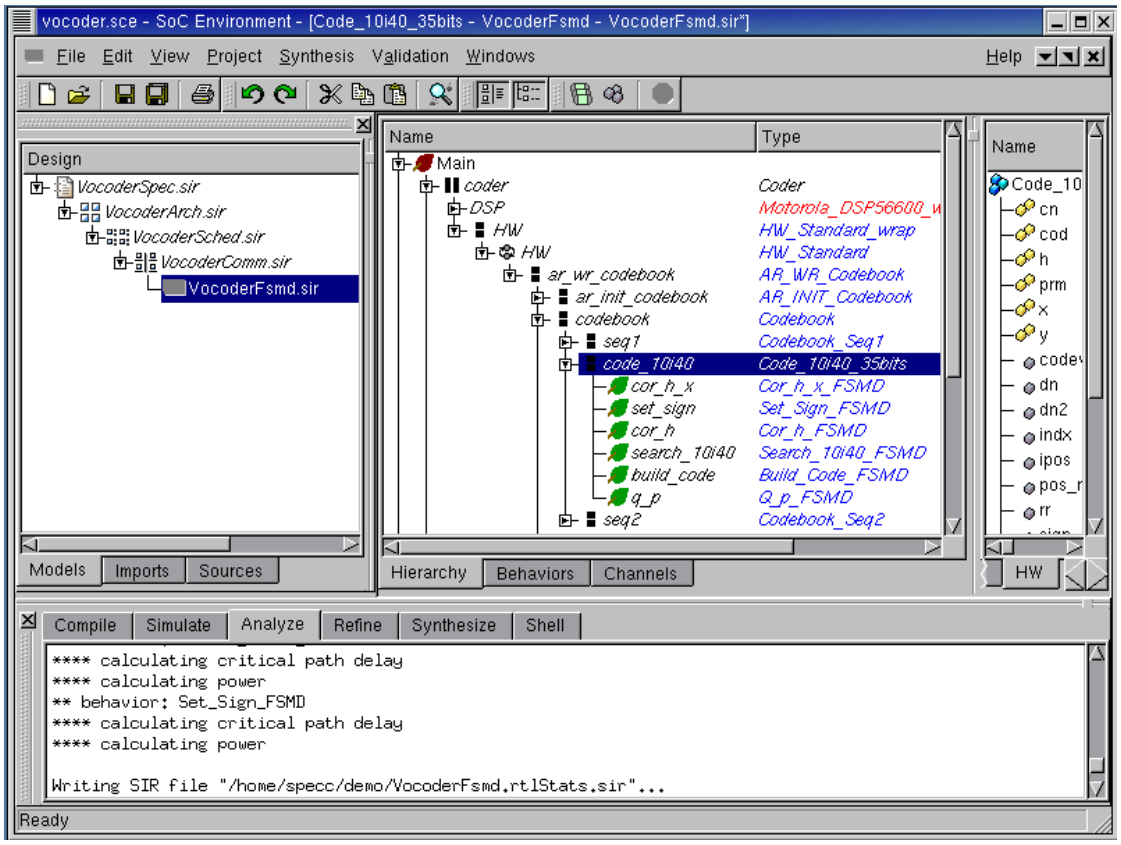
We are now done with RTL component allocation. Left click on OK to save the allocation information in the model.

4.3.3.5. Analyze allocated SFSMD model



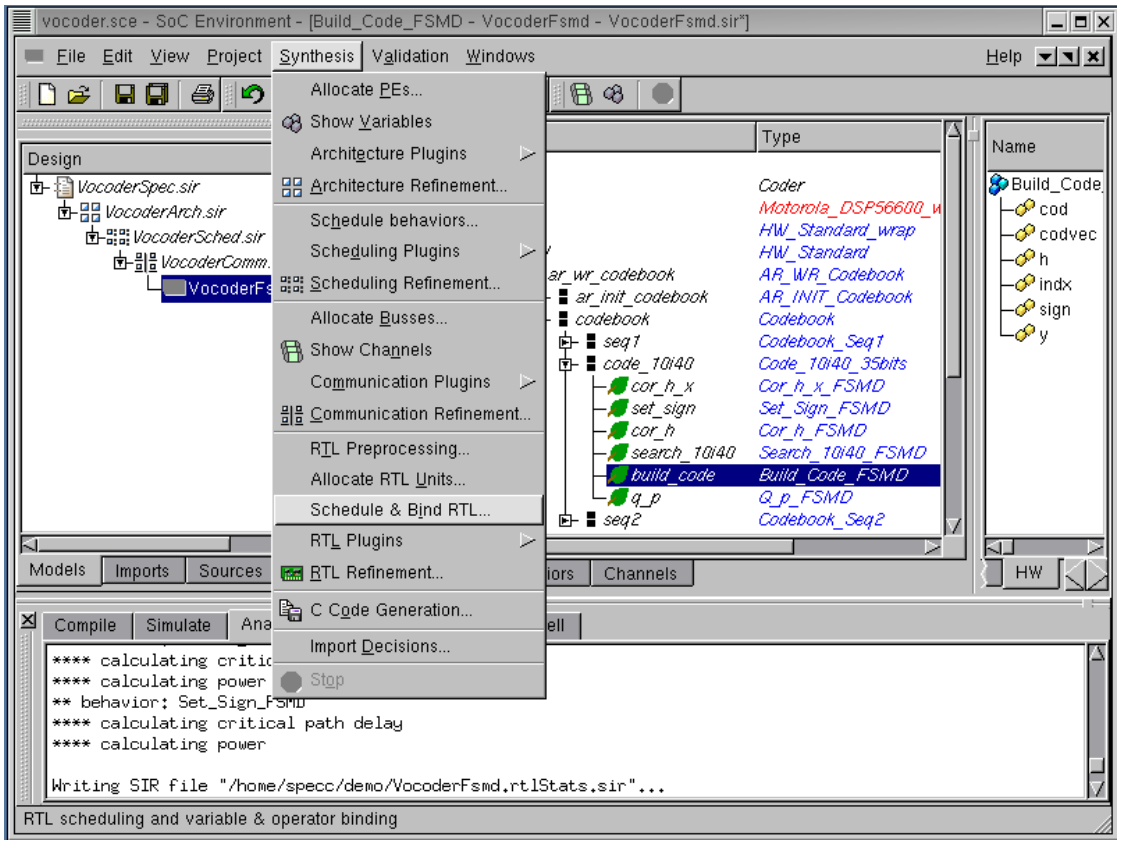
Before scheduling and binding, we may check how RTL allocation will affect performance, area, and power in the design. To do so, we can go over RTL analysis again. we select the behavior "Code_10i40_35bits", for which we want to get the statistical information. The RTL analysis is performed by selecting Validation⟶Analyze RTL from the menu bar.

4.3.3.6. Analyze allocated SFSMD model (cont'd)



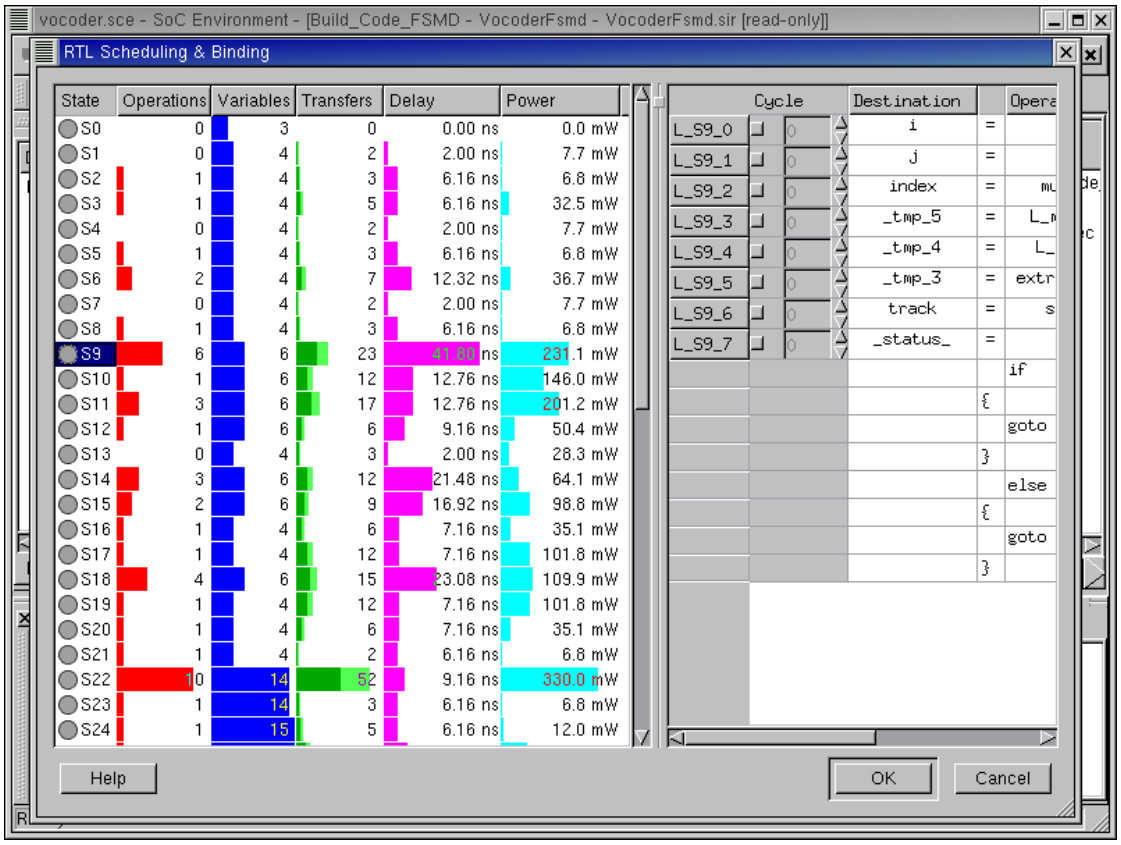
RTL analysis tool will go over all sub-behaviors in behavior "Code_10i40_35bits", and generate the more accurate statistical information with the help of allocation information.

4.3.3.7. Analyze allocated SFSMD model (cont'd)



Now, we will look at RTL analysis result because we allocated RTL components for the design by selecting Synthesis⟶Schedule & Bind RTL from the menu bar. Choose the behavior "Build_Code_FSMD" from the hierarchy.

4.3.3.8. Analyze allocated SFSMD model (cont'd)



The RTL Scheduling & Binding window pops up showing all the states in the behavior "Build_Code_FSMD". In the left-most columns, we can see the estimated delay and powers for each state. For example, state S9 will take 41.80 ns to execute and consume 180.0 mW.

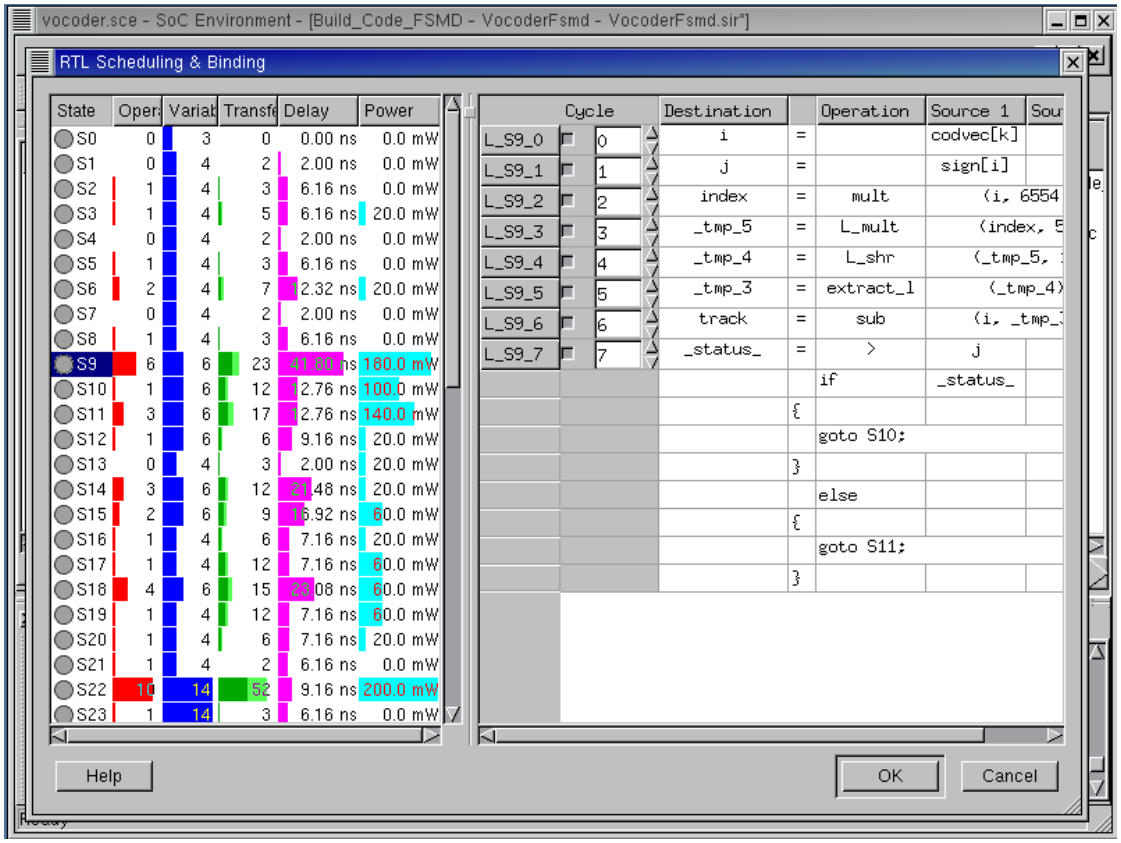
4.4. RTL Scheduling and Binding

The most important steps during RTL synthesis are scheduling and binding. Scheduling is to decide the start time of operations in a design. Binding is to map operations to functional units (function binding) and to map variables to storage units (storage binding), and to map data transfers to buses (connection binding). Due to the interdependence of scheduling and binding, the order of these steps may be interchanged to get better design.

In our RTL design methodology, we provide manual scheduling and binding for the designers to make decision for scheduling and binding. But manual scheduling and binding takes too much time for the designers to do and is tedious and error-prone task. We will provide automatic scheduling and binding tools by RTL plugins.

Note that if reader is not interested in how to do manual scheduling and binding, she or he can skip this section to go directly Section 4.4.2 *Schedule and bind automatically* (page 200).

4.4.1. Schedule and bind manually (optional)



SCE allows for the designer to manually schedule and bind the operations. However, this is a tedious task and can be done by automated tools. To perform automatic scheduling and binding, the designer can skip the manual step and go directly to Section 4.4.2 *Schedule and bind automatically* (page 200).

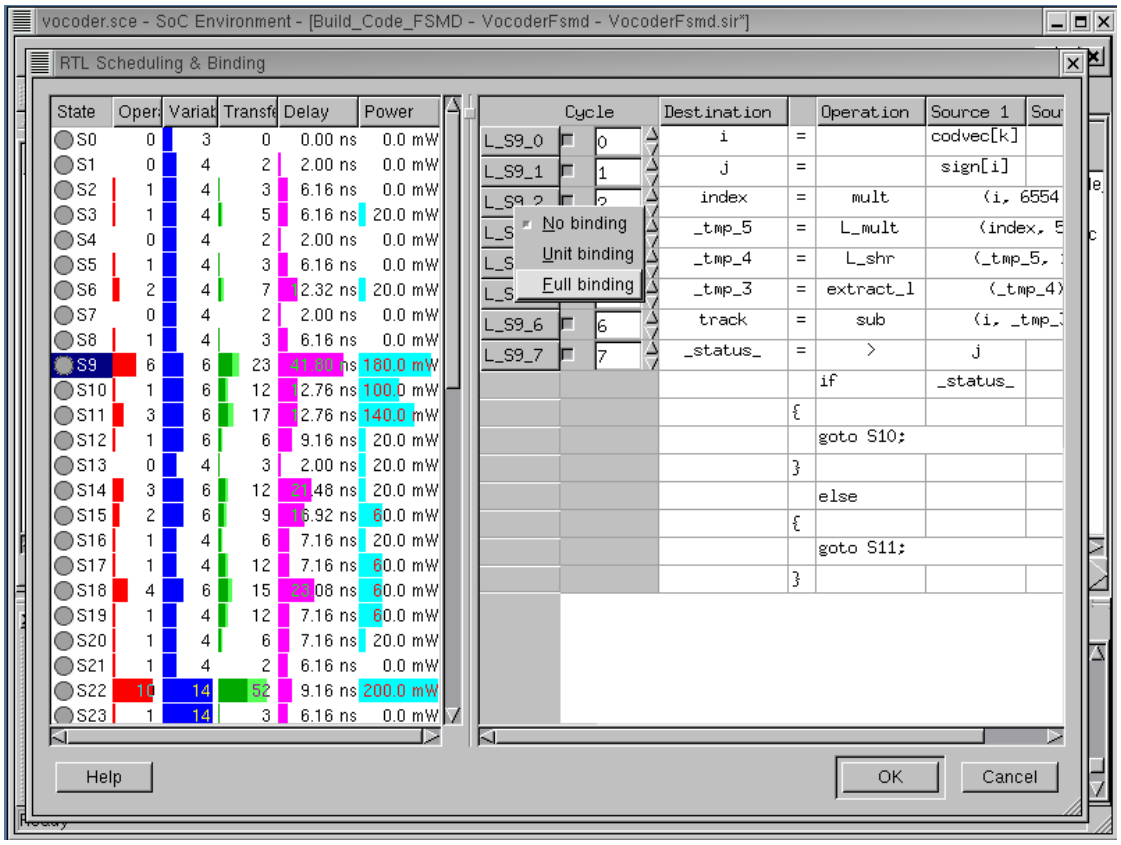
If RTL Scheduling & Binding window is not open yet, we have to open it again by selecting Synthesis⟶Schedule & Bind RTL from the menu bar. Choose the behavior "Build_Code_FSMD" from the hierarchy.

we will show how to specify control step for each statement in a state. In RTL Scheduling and Binding window, we select "S9" to do manual scheduling and binding. In the right side panel of the RTL Scheduling & Binding window, left click on the right side of the label "L_S9_0". This activates the Cycle column for "L_S9_0". We can specify the control step for it. In this way, we can specify control step for all statement in the state S9.

Note that if reader is not interested in how to do manual schedul-

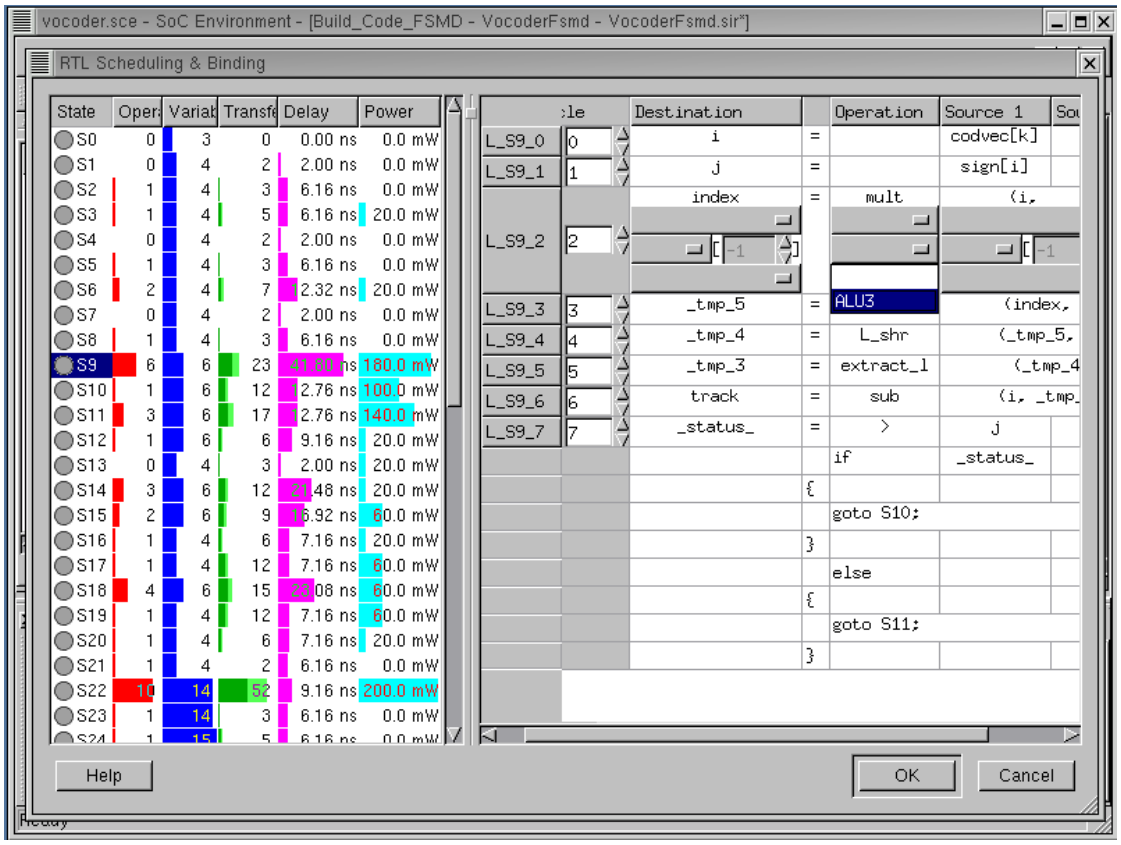
ing and binding, she or he can skip this section to go directly Section 4.4.2 *Schedule and bind automatically* (page 200).

4.4.1.1. Schedule and bind manually (optional) (cont'd)



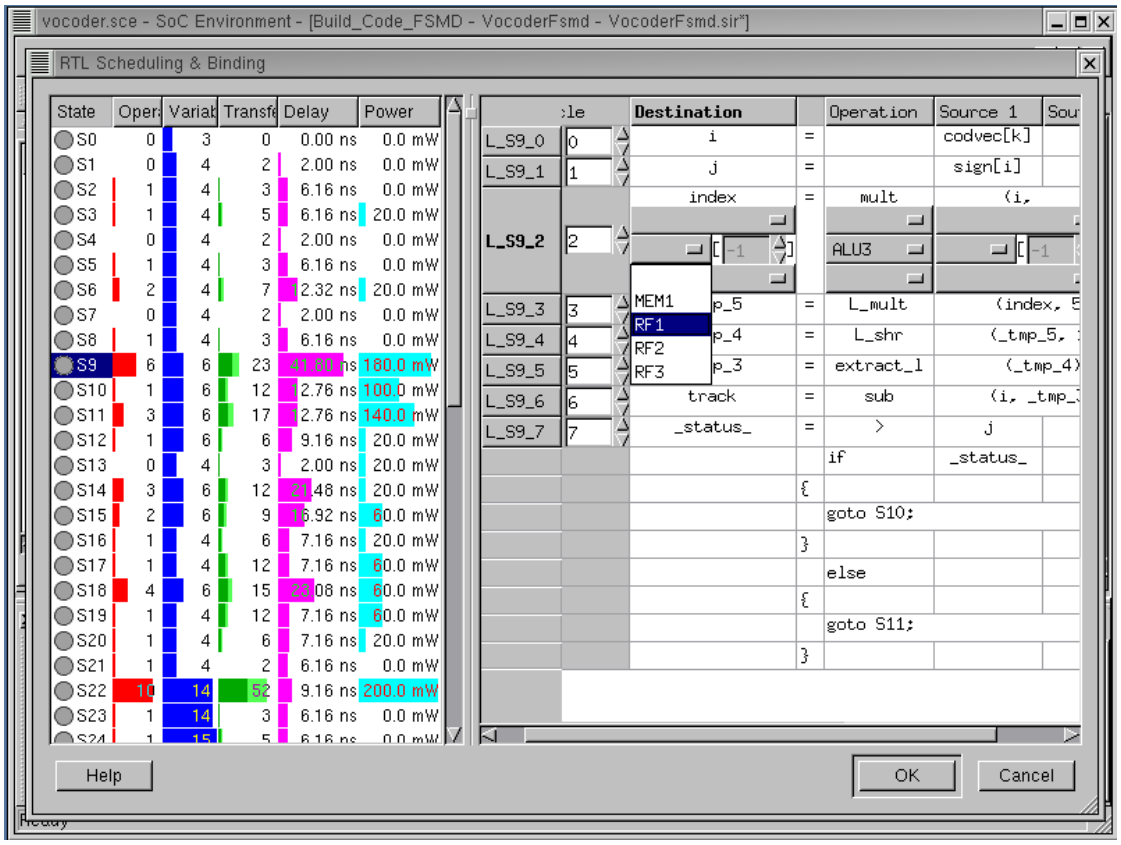
To perform manual binding for operations in the state S9, right click on Label, "L_S9_2". It will pop up a menu for the binding options. Select Full binding.

4.4.1.2. Schedule and bind manually (optional) (cont'd)



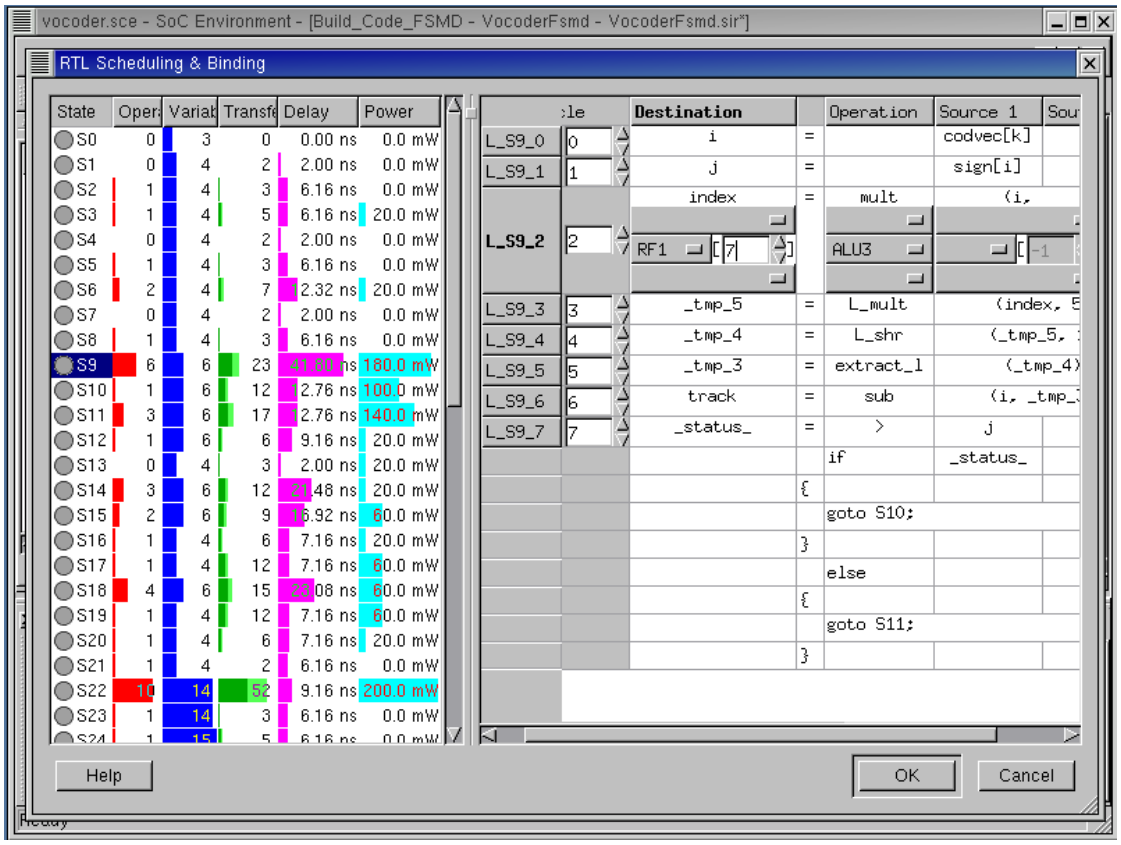
Each column in a statement in right side of the window is now expanded to allow manual binding. We will bind a function call, mult (Operation column), to "ALU3". To do so, left click on 2nd blank row of the Operation column. Then pull-down menu pops up and shows all functional units which can perform function call mult. In this case, one possible functional unit, "ALU3" is shown in the pull-down menu.

4.4.1.3. Schedule and bind manually (optional) (cont'd)



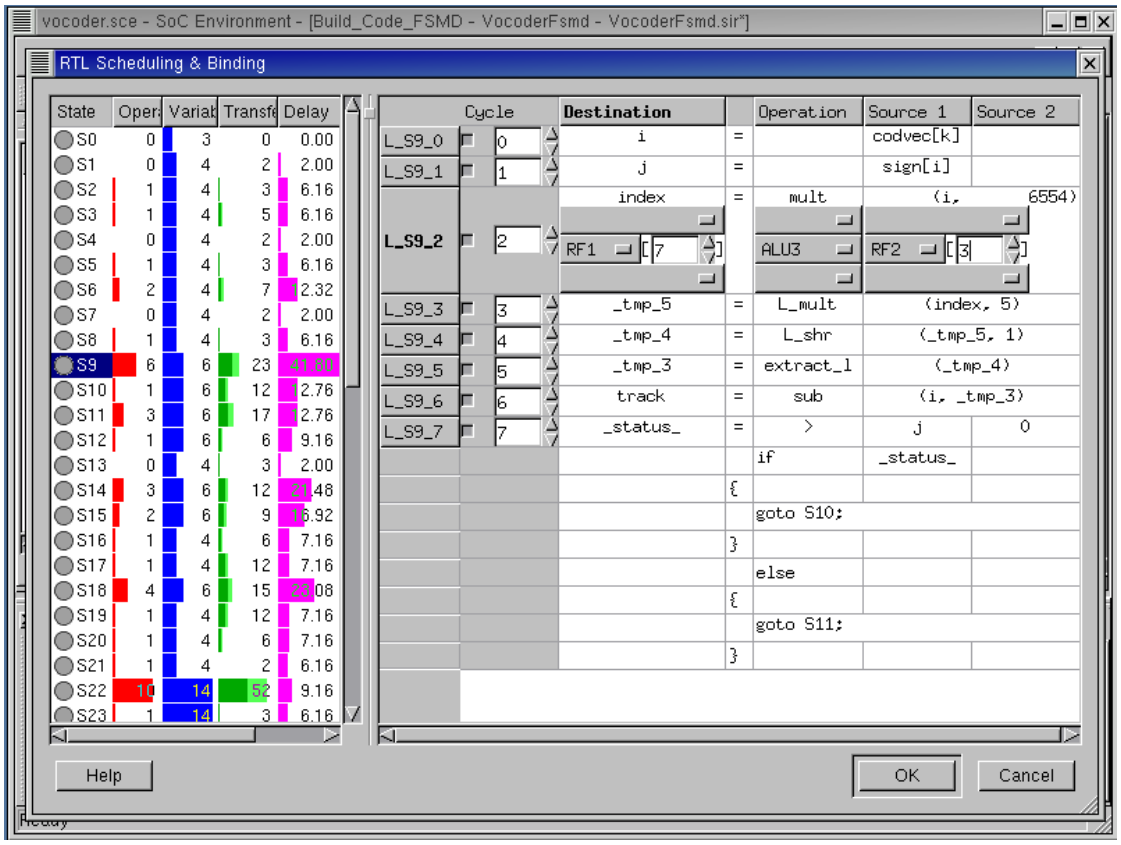
We will bind a target variable index (Destination column) to RF1[7]. To do so, left click on 2nd blank row of the Destination column. Then pull-down menu pops up and shows all storage units In this case, four storage units such as "MEM1", "RF1", "RF2" and "RF3" are shown in the pull-down menu. Click on "RF1" to select "RF1".

4.4.1.4. Schedule and bind manually (optional) (cont'd)



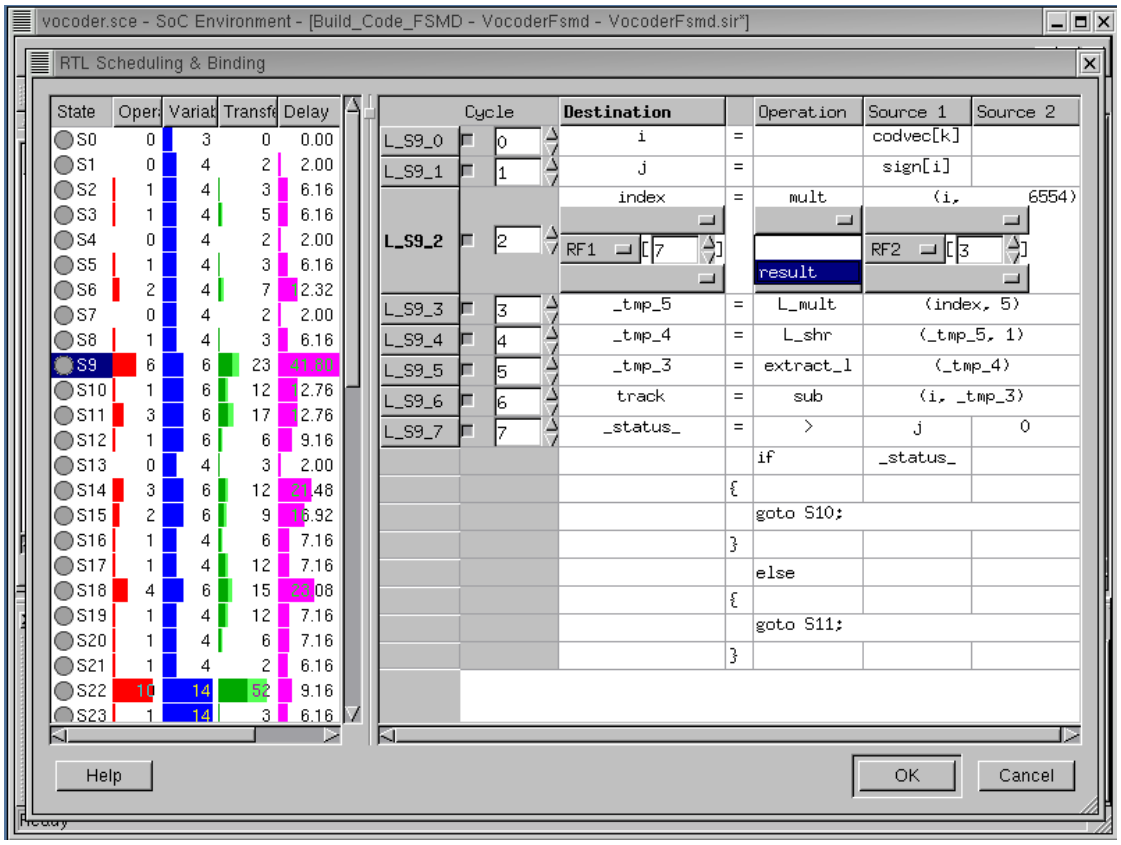
For storage unit binding, the address of the variable in the memory should be specified. Left click on right side of the 2nd row of Destination column. Specify the memory address to 7 for variable "index". The -1 in address field for a memory is default value which means that the address for the memory is not bound yet.

4.4.1.5. Schedule and bind manually (optional) (cont'd)



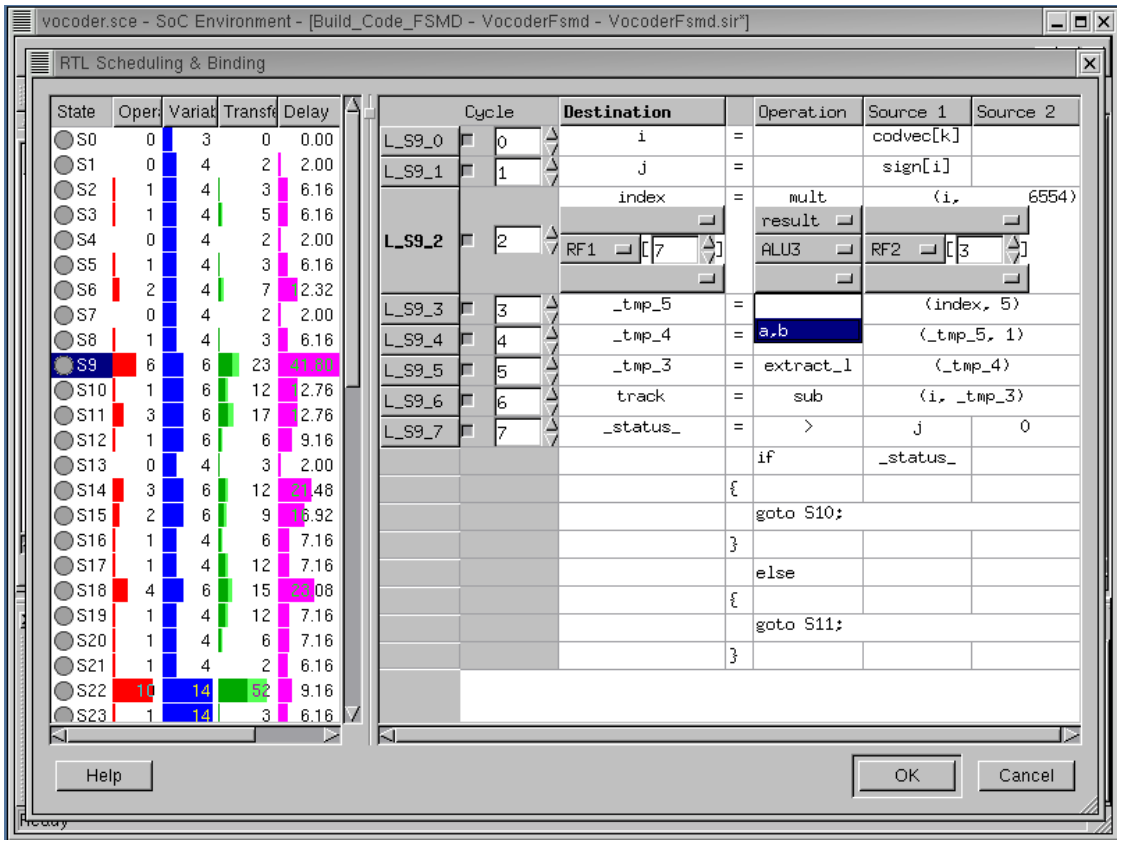
Likewise, source variable, "i" is bound to RF2[3].

4.4.1.6. Schedule and bind manually (optional) (cont'd)



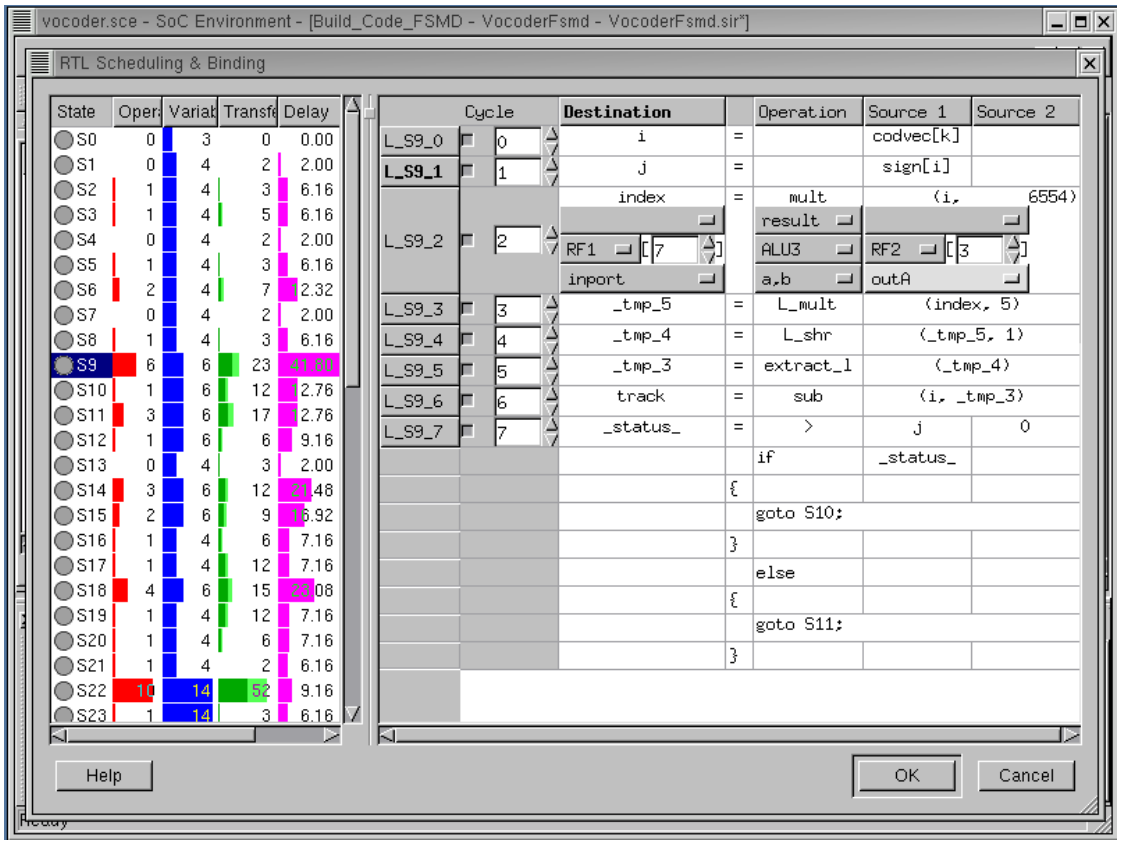
So for, we performed functional unit and storage unit binding. We can specify more information on binding, such as ports of the functional unit and storage unit and buses for data transfers. For the output port binding of the functional unit, left click on the 1st row of the Operation column which will show all output ports in ALU3 unit.

4.4.1.7. Schedule and bind manually (optional) (cont'd)



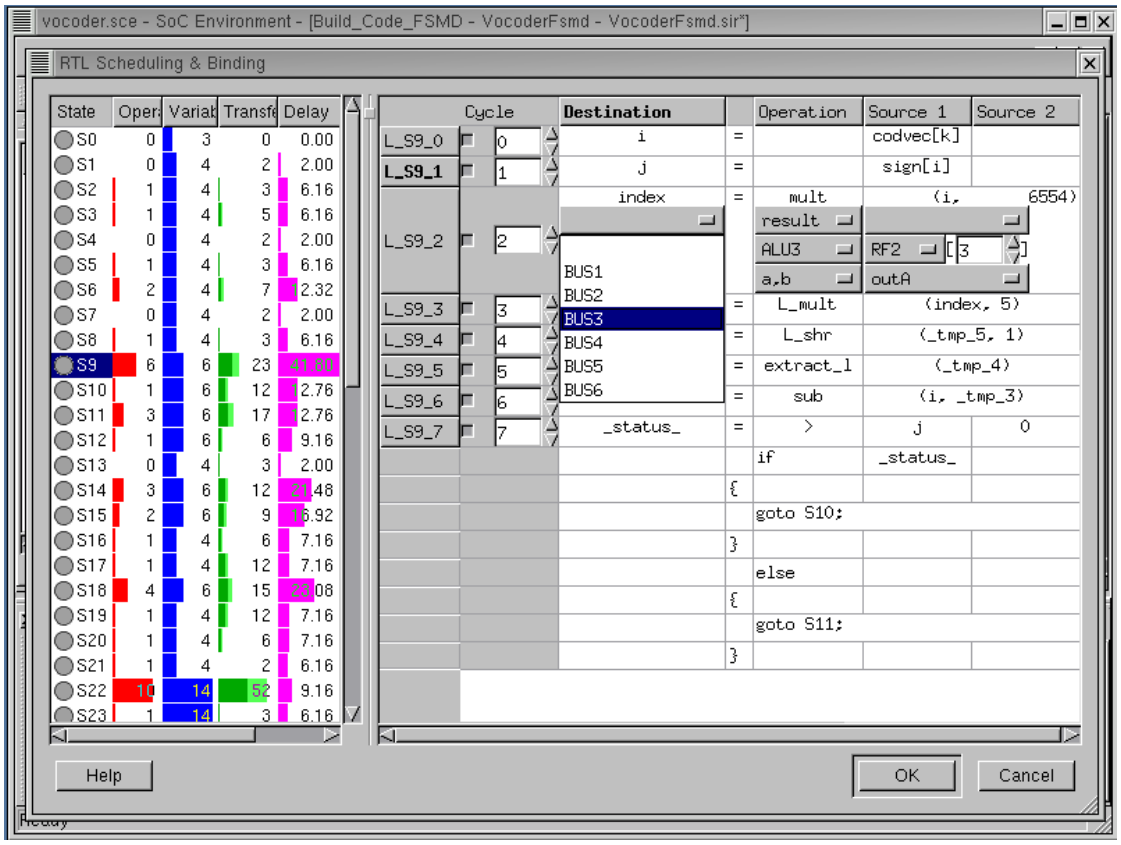
For the input port binding of the functional unit, left click on the 3rd row of the Operation column which will shows all input ports in ALU3 unit.

4.4.1.8. Schedule and bind manually (optional) (cont'd)



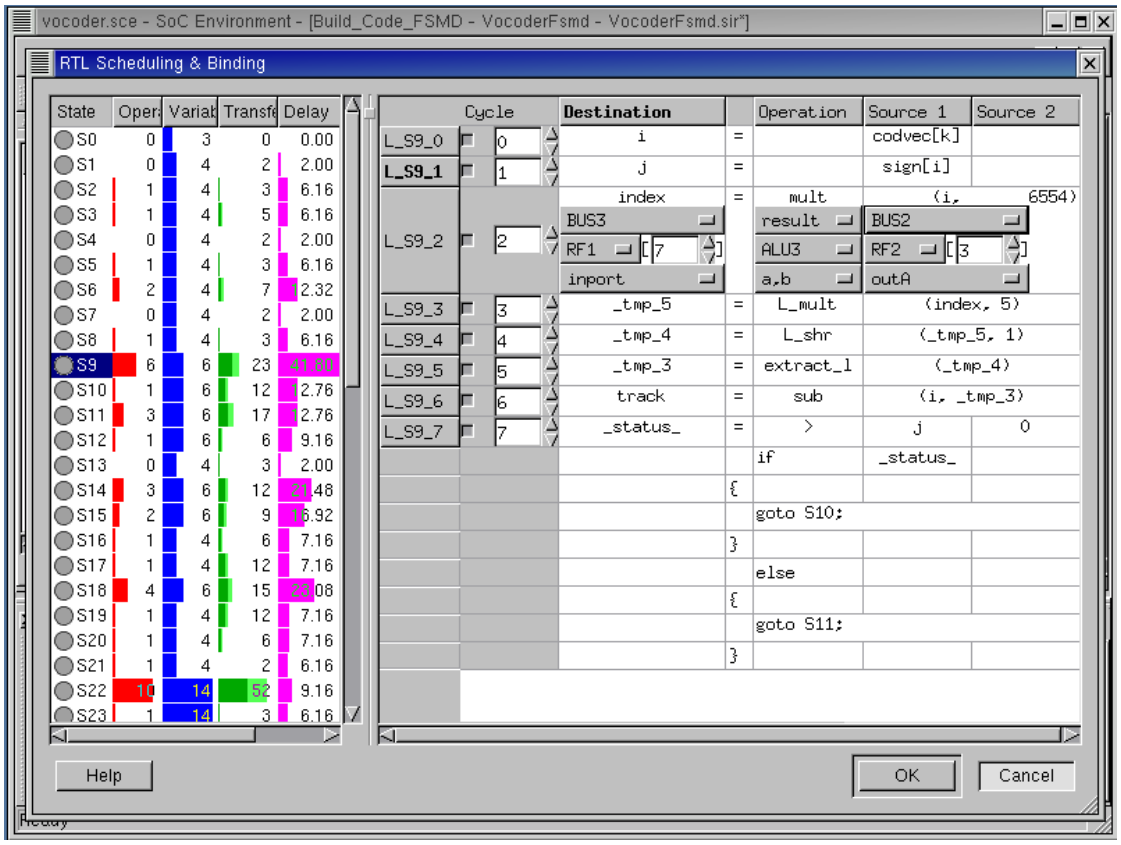
In this way, we can select write port for the write storage unit (RF1[7]) and read port for the read storage unit (RF2[3]).

4.4.1.9. Schedule and bind manually (optional) (cont'd)



For the bus binding, left click on the 1st row of the Destination column which shows all allocated buses. For target variable "index", "BUS3" is selected for write.

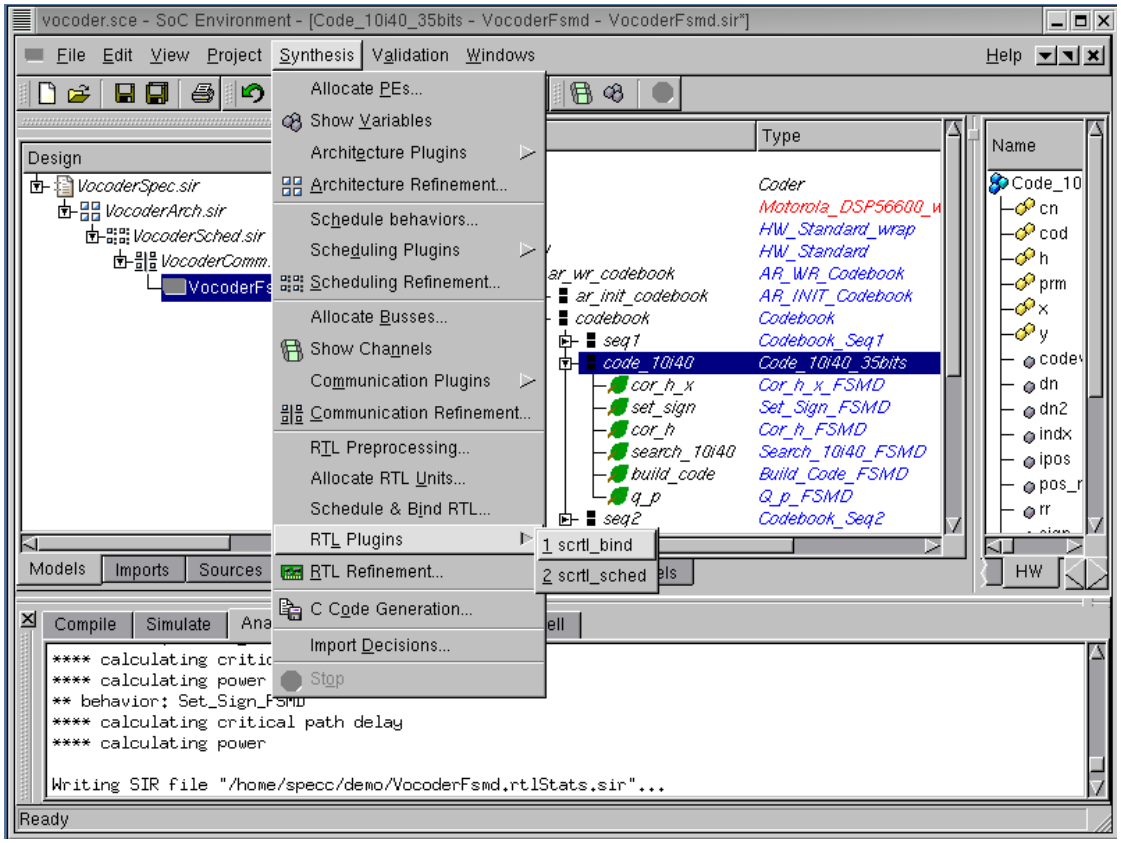
4.4.1.10. Schedule and bind manually (optional) (cont'd)



In this way, we can perform all binding in the RTL Scheduling and Binding window. However, manual binding takes too much time and is an error-prone task. An easier alternative is to use automatic scheduling and binding tools.

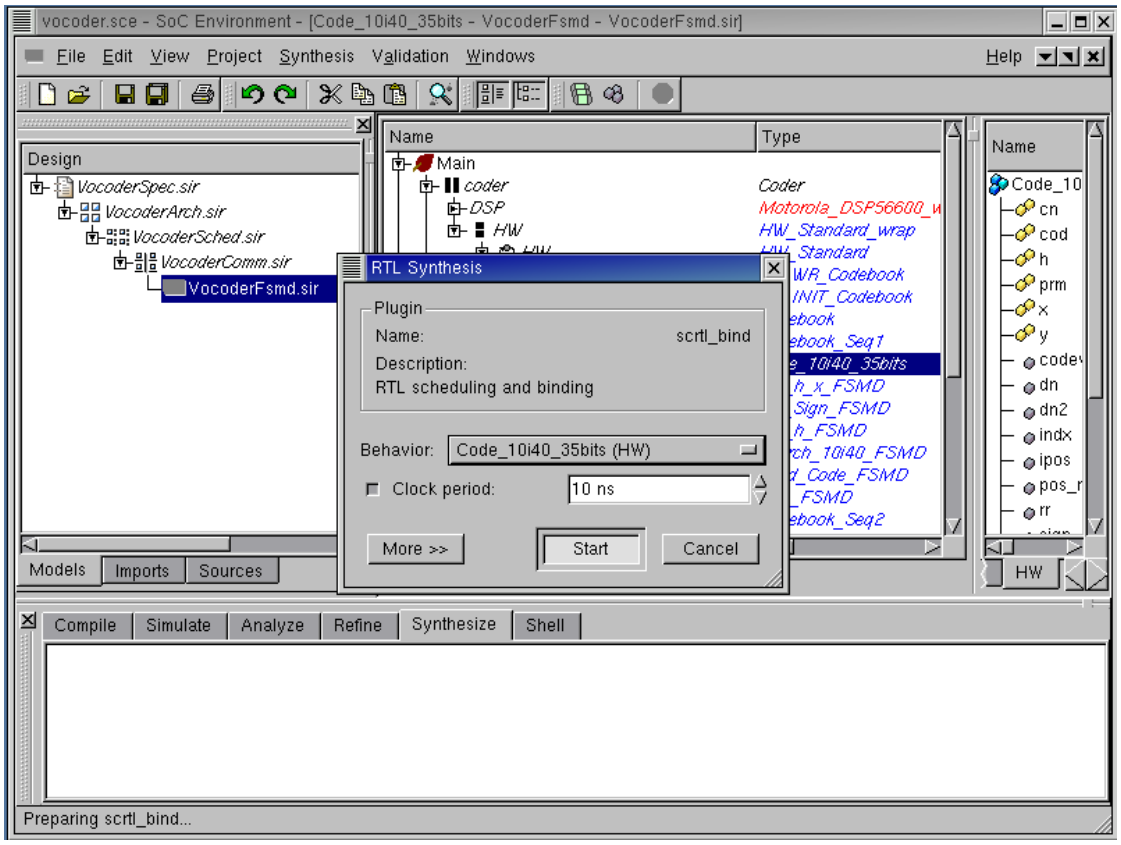
Left click on Cancel. Otherwise, the scheduling and binding information will be inserted and then used by automatic scheduling and binding tools. It may generate an incorrect RTL model.

4.4.2. Schedule and bind automatically



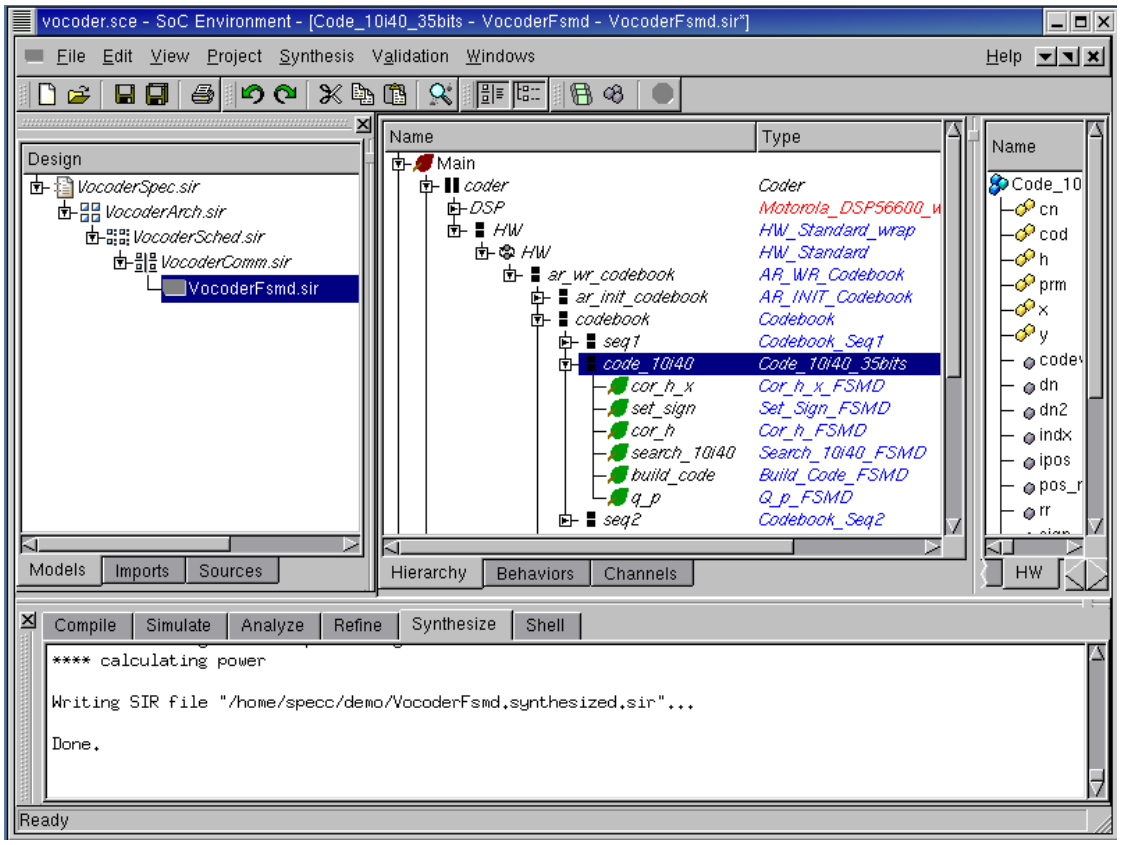
As already discussed, manual scheduling and binding takes too much time for a designer to do and also is an error-prone task. We will now perform scheduling and binding with the help of tools which implement scheduling and binding algorithms. In our design flow, an automatic decision making tool for system-level design is called a "Plug in". For RTL scheduling and binding, we call "RTL Plugins" by selecting Synthesis⟶RTL PlugIns⟶scrtl_bind from the menu bar. Before that, we have to select a behavior "Code_10i40_35bits".

4.4.2.1. Schedule and bind automatically (cont'd)



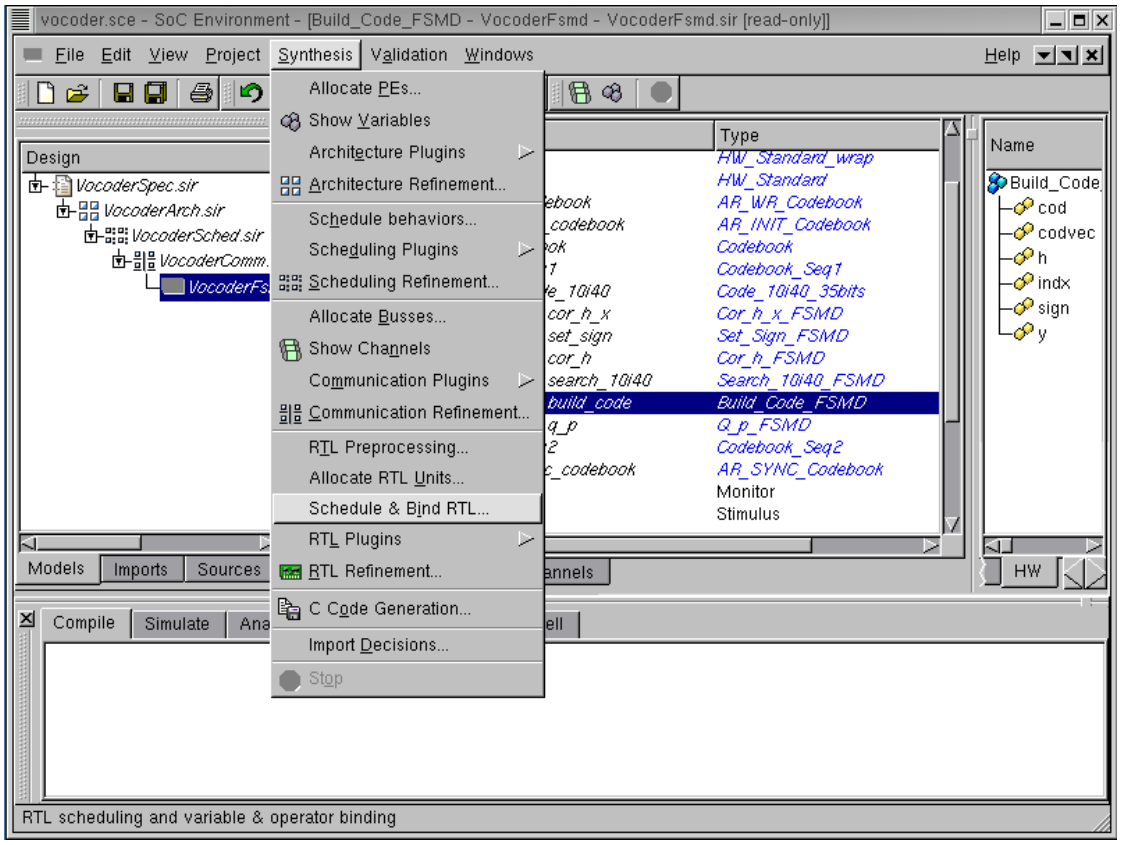
An RTL Synthesis dialog box pops up. In the middle of the dialog box, a pull-down list is available to select the desired behavior. The default behavior in the list is the one that is highlighted in the behavior hierarchy tree. For our demo, select behavior "Code_10i40_35bits (HW)" from the list. By default, the clock period of the behavior is 10 ns. Now click on Start to begin "scrtl_bind".

4.4.2.2. Schedule and bind automatically (cont'd)



Note that "scrtl_bind" annotates scheduling and binding information into SFSMDs for all 6 sub-behaviors of the behavior "Code_10i40_35bits", as seen in the logging window. The tool finally generates the SFSMD model for the behavior "Code_10i40_35bits".

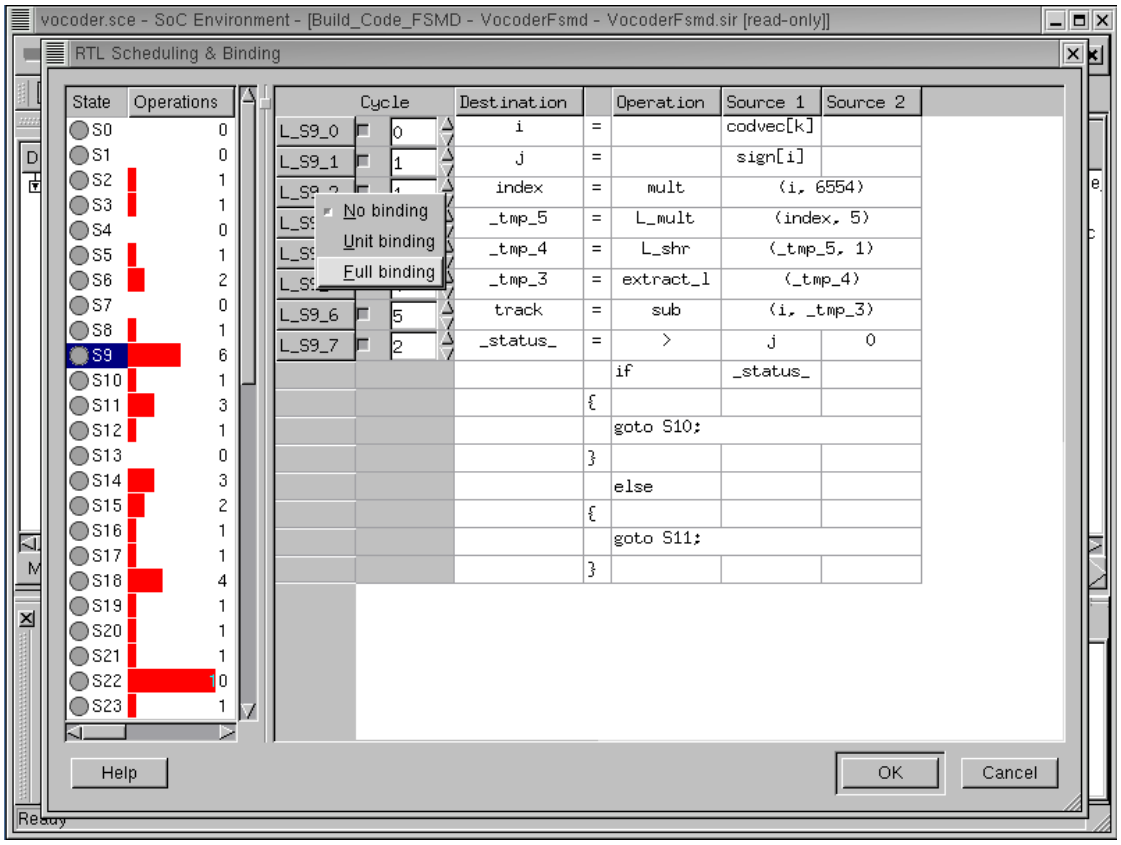
4.4.2.3. Browse scheduling and binding result (optional)



To check the scheduling and binding result generated by "scrtl_bind", we have to go over to RTL Scheduling & Binding window again by selecting Synthesis⟶RTL Scheduling & Binding from the menu bar. Before that, we have to select a behavior "Build_Code_FSMD".

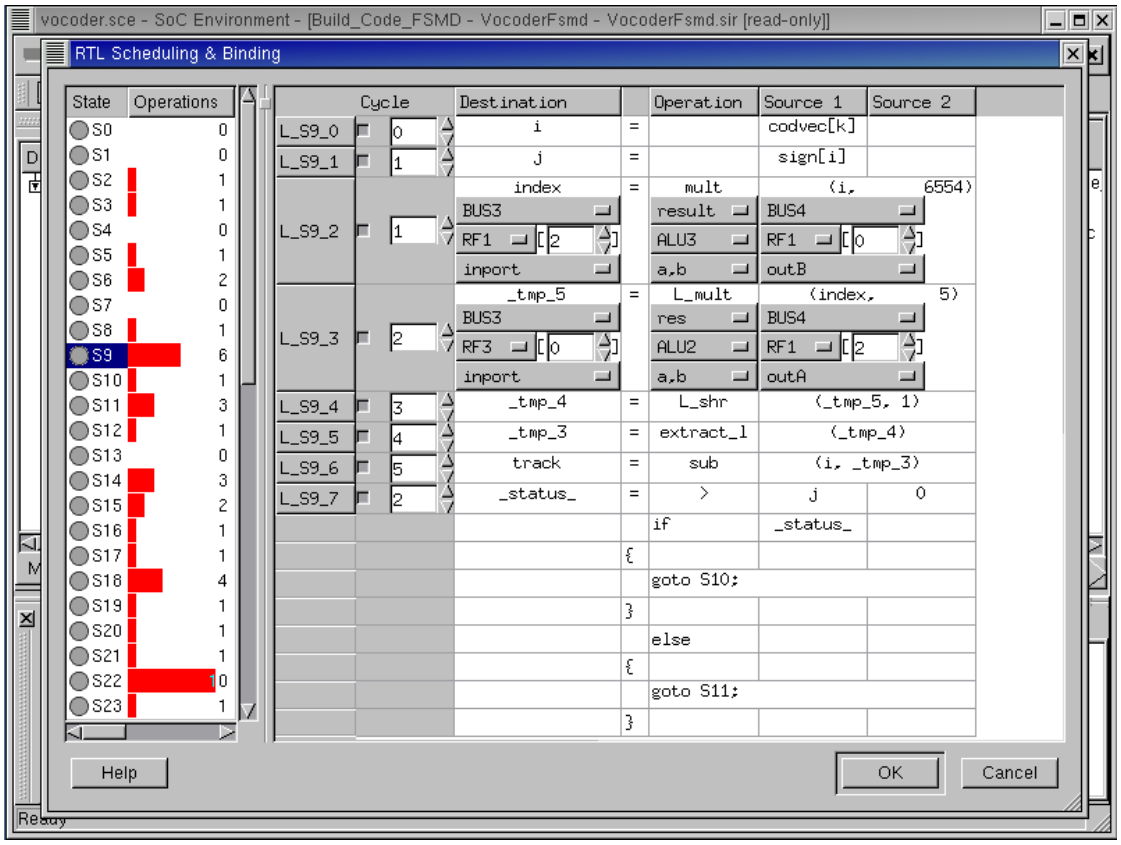
If the reader is not interested in details of the scheduling and binding results, she or he skips this section and go directly Section 4.5 *RTL Refinement* (page 206).

4.4.2.4. Browse scheduling and binding result (optional) (cont'd)



In the RTL scheduling and Binding window, Cycle column shows the control step of each statement. To see the binding information, we activate Full binding by selecting Full binding in the binding pop-up menu.

4.4.2.5. Browse scheduling and binding result (optional) (cont'd)



This is the scheduling and binding result for the L_S9_2 and L_S9_3 statement. The statement L_S9_2 is scheduled control step 1 relative to the start of state S9. The function call "mult" is performed by ALU3. The variable "index" in statement L_S9_2 is bound to RF1[2] which stores the result of the function call "mult" through the bus "BUS3".

Left click on Cancel.

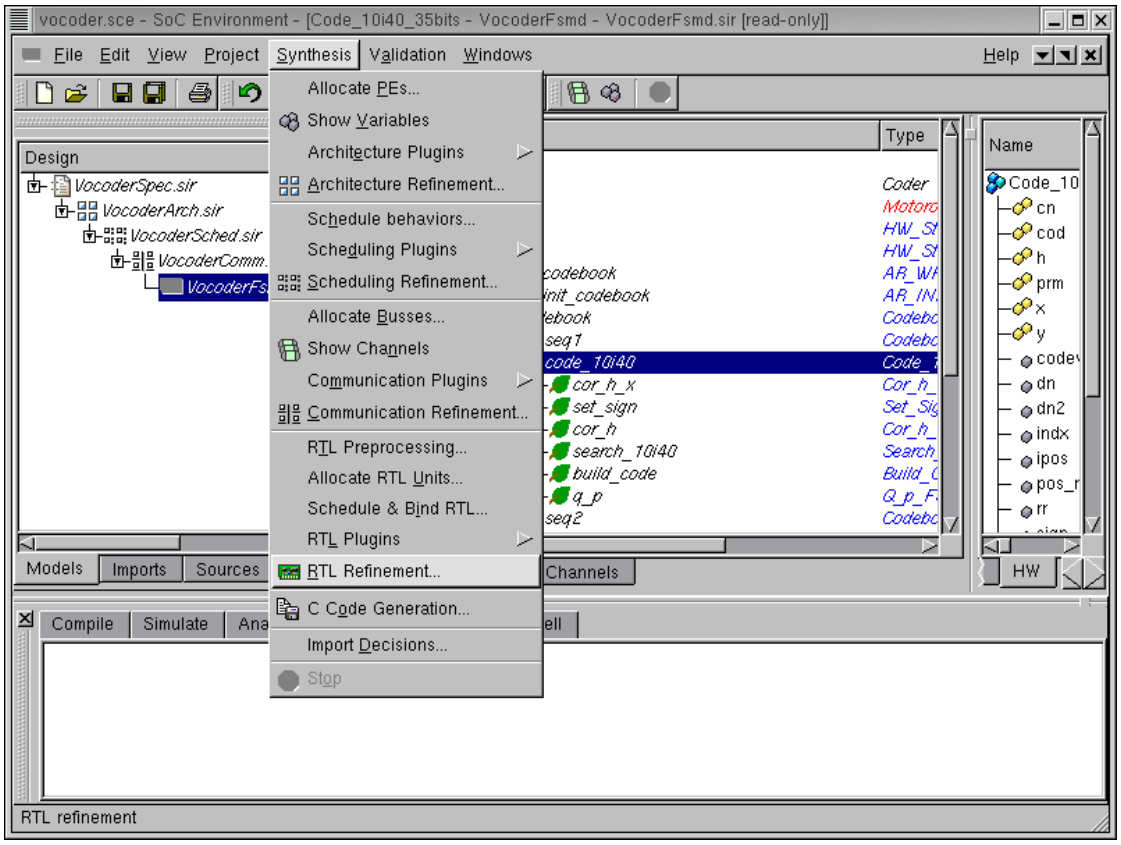
4.5. RTL Refinement

So far, we performed allocation, scheduling and binding of which information is annotated into the SFSMD model. Then the SFSMD model should be refined into cycle-accurate RTL model, which is represented by finite state machine with data (FSMD). The cycle-accurate model will reflect all scheduling and binding information.

Basically, this step will split the state to the multiple states reflecting scheduling information. Now each state will take exactly one clock period to perform.

The RTL refinement tool can generate cycle-accurate FSMD models in various hardware description languages such as Verilog HDL and Handel-C in addition to SpecC. The Verilog HDL model will be used as input for commercial logic synthesis tool like Design Compiler from Synopsys. Also the Handel-C model will be fed into Celoxica Design Kit to generate gate-level netlist.

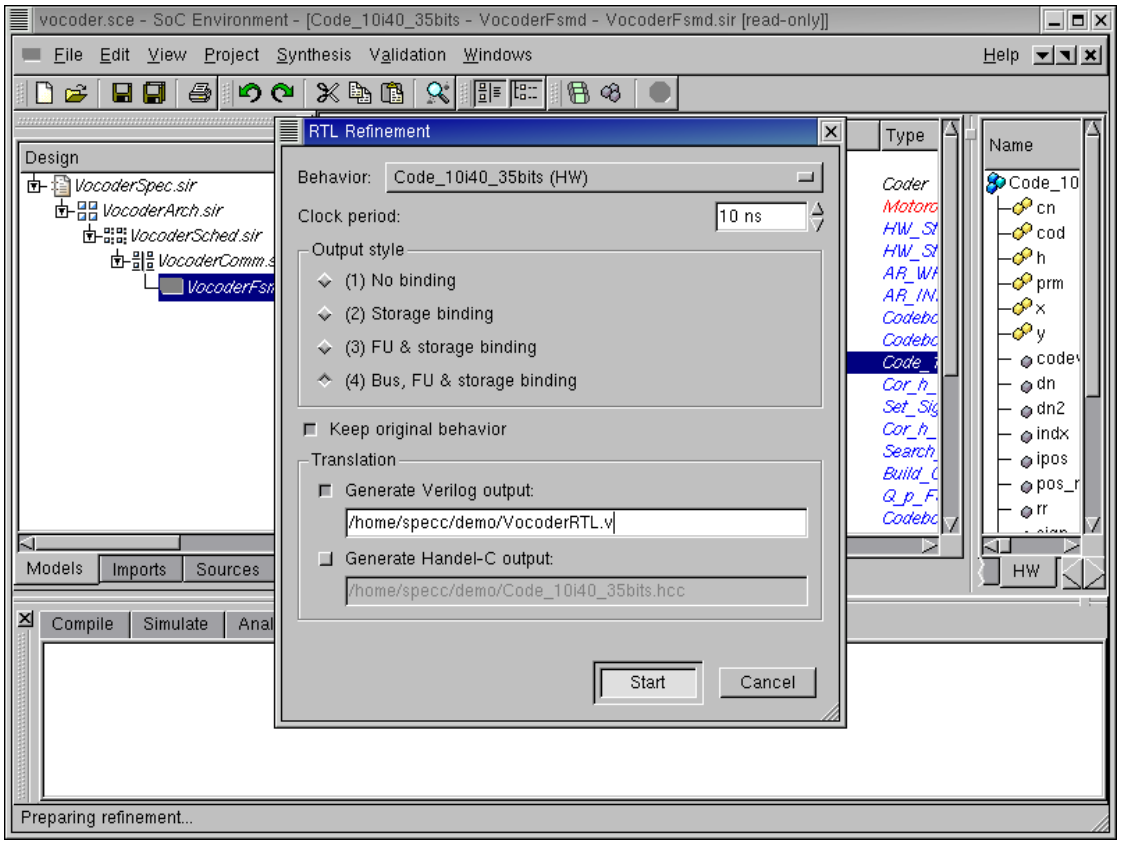
4.5.1. Generate RTL model



We refine the SFSMD model to a cycle-accurate model by selecting Synthesis⟶RTL Refinement from the menu bar.

The refinement step will split the state into multiple states reflecting the scheduling information. Also, each state will take exactly one clock period to execute.

4.5.1.1. Generate RTL model (cont'd)



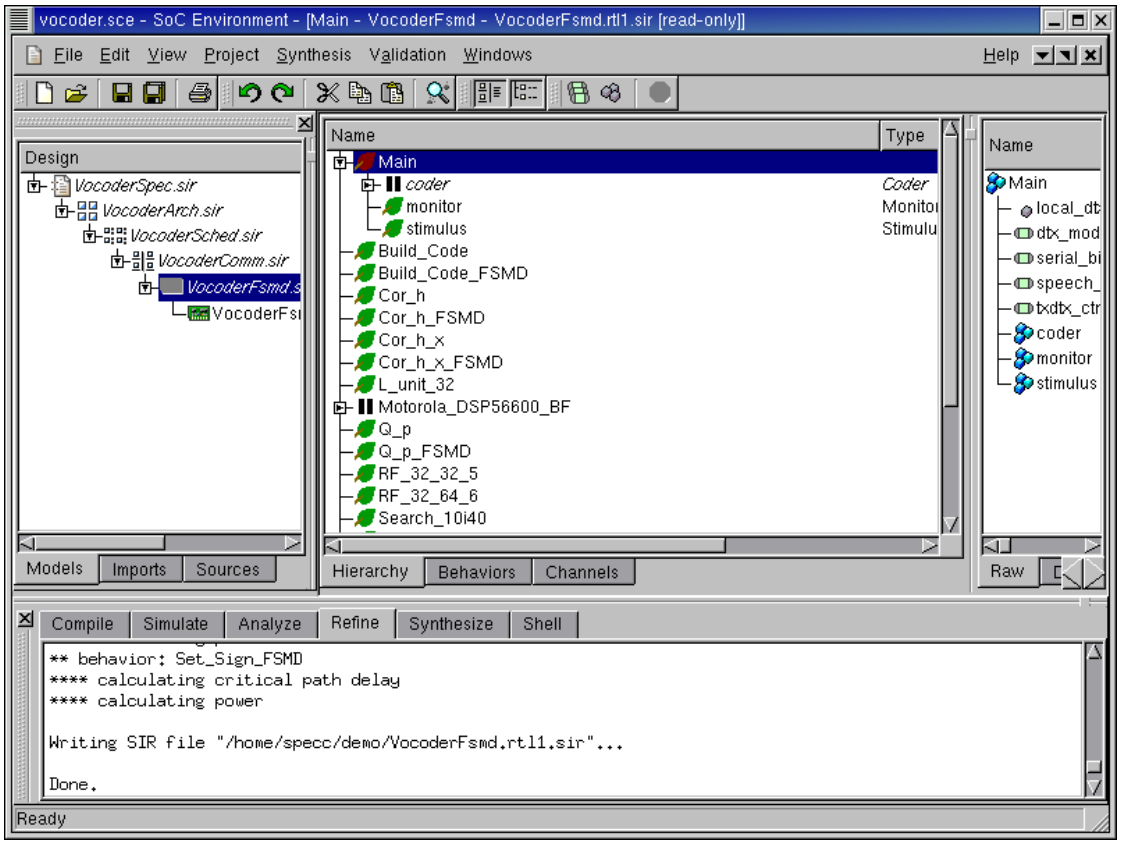
The RTL Refinement dialog box pops up showing us all options which can be used for the refinement tool. At the top of the dialog box, a pull-down list is available to select the desired behavior to be refined. The default behavior is the one that is highlighted in the behavior hierarchy tree. For our demo, select "Code_10i40_35bits (HW)" from the list then left click on Start to begin RTL refinement. Notice that like in the earlier refinement phases, we have options for partial refinement steps. The user might avoid some binding steps if he wants to look at intermediate models. Also note that we have selected a clock period of 10 ns, corresponding to the speed of our custom hardware unit. It may be recalled that while selecting the hardware component, we specified a hardware component with clock speed of 100 Mhz, which imposes a clock period of 10 ns.

The RTL refinement tool can generate cycle-accurate FSMD model in various hardware description language such as Verilog HDL and Handel-C in addition to SpecC. The Verilog HDL model will be used to be input of the commercial logic synthesis tools such as Design Compiler from Synopsys. Also the Handel-C model will be fed into Celoxica Design Kit to generate gate-level netlist. In this demo, we will generate SpecC

RTL and Verilog HDL model for the design.

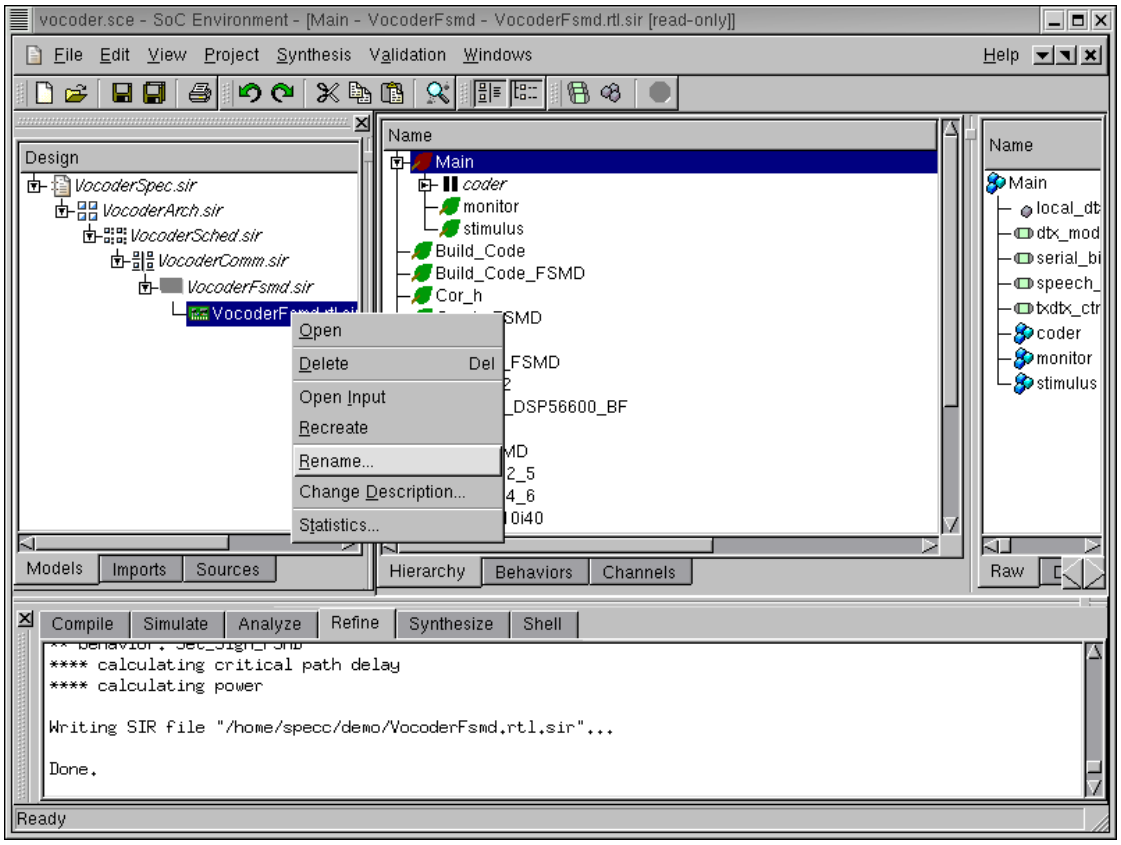
Change the output file name for the Verilog HDL model to "VocoderRTL.v".

4.5.1.2. Generate RTL model (cont'd)



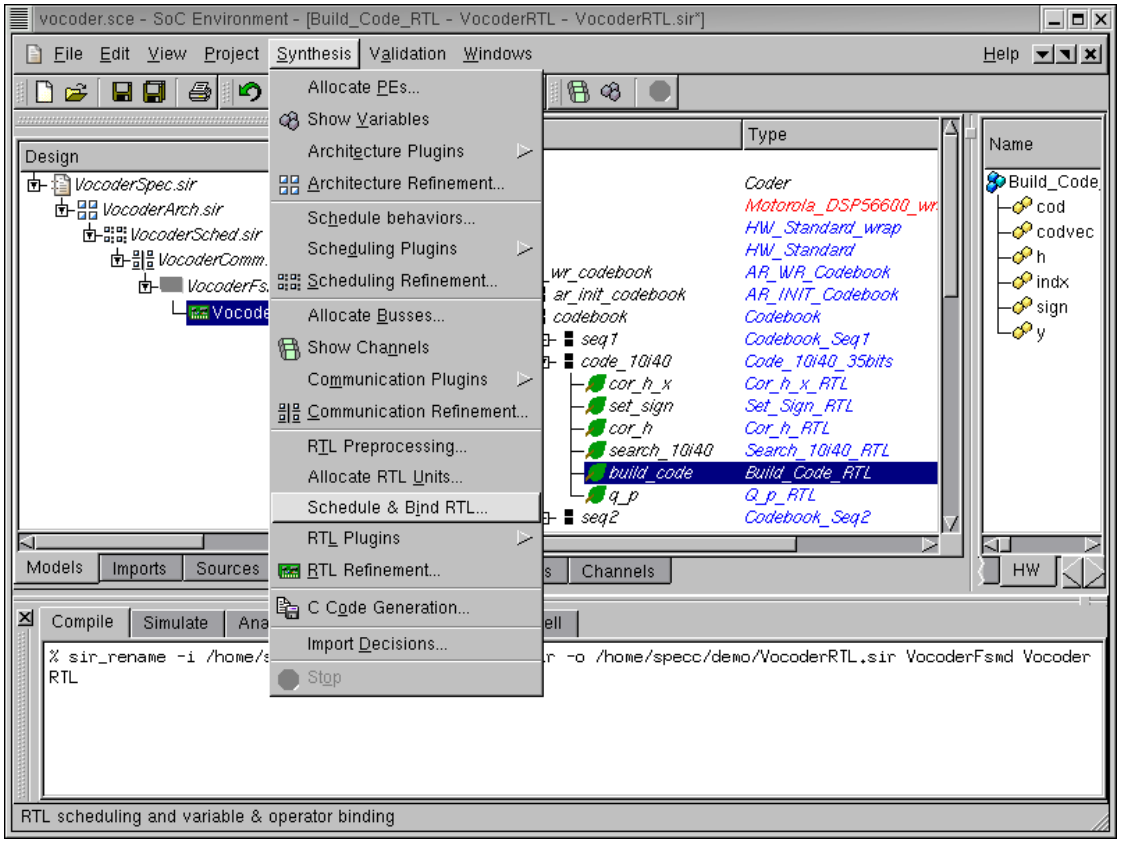
Note that the RTL refinement step generates a new RTL model for 6 sub-behaviors of the behavior "Code_10i40_35bits", as seen in the logging window. Also note that a new model "VocoderFsmd.rtl.sir" is added in the Project manager window.

4.5.1.3. Generate RTL model (cont'd)



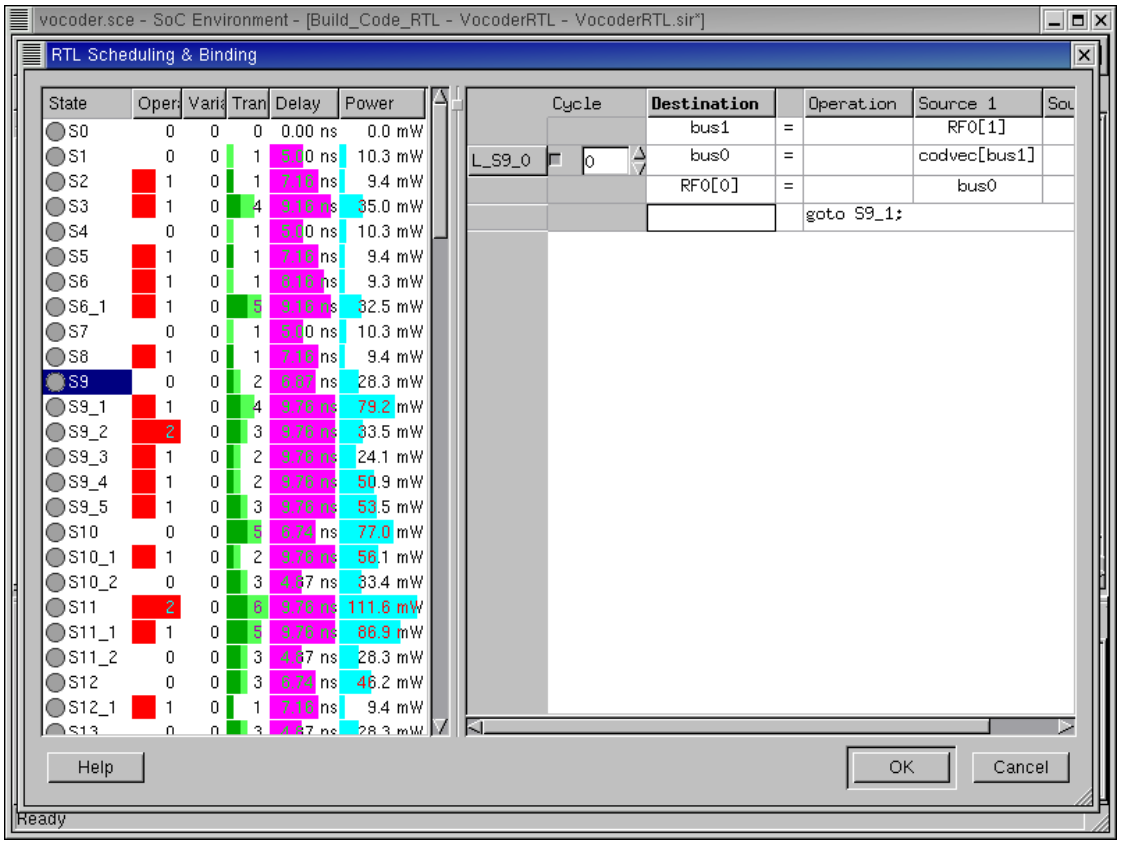
Like before, we must give our new model a suitable name. We can do this by right clicking on "VocoderFsmd.rtl.sir" and selecting Rename from the pop up menu. Rename the model to "VocoderRTL.sir".

4.5.2. Browse RTL model



In order to look at RTL model for the behavior "Build_Code_RTL", select Synthesis⟶Schedule & Bind RTL from the menu bar.

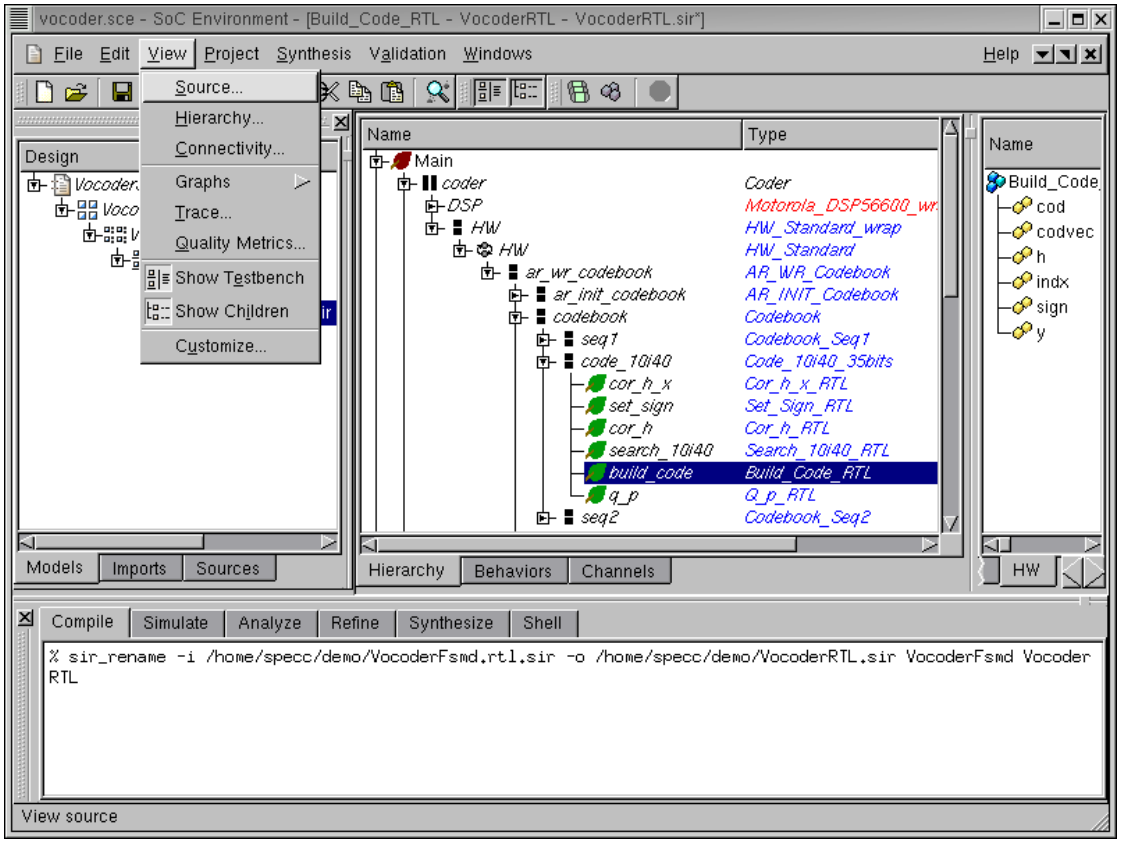
4.5.2.1. Browse RTL model (cont'd)



In the right-most column of the RTL Scheduling and Binding window, some states are split to multiple states. For example, state S9 is split to 6 states, S9, S9_1, ..., S9_5. Note that the delay of these states is less than 10 ns in Delay in the right-most column.

Left click on Cancel.

4.5.3. View RTL model (optional)

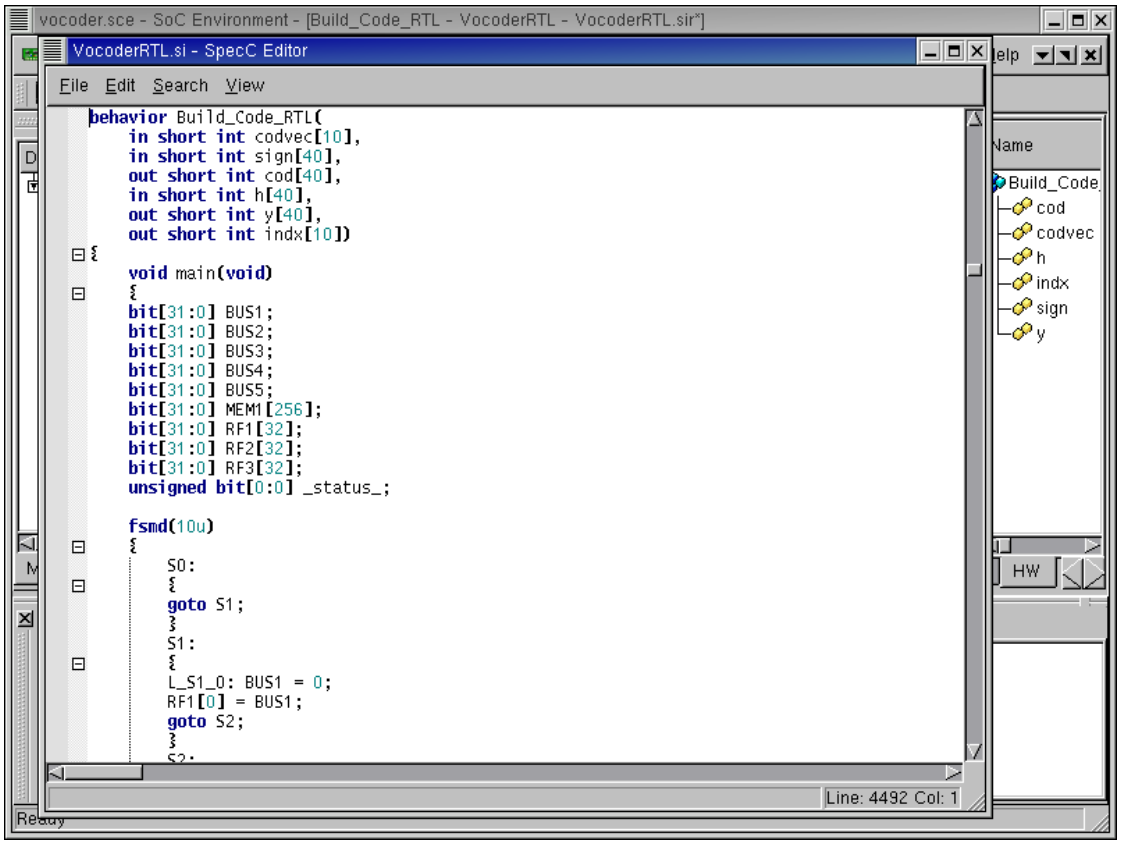


We now browse through the newly created model in the Design hierarchy window. Note that the type of the instance "build_code" has now changed to "Build_Code_RTL" after RTL refinement.

Select the behavior "Build_Code_RTL" by left clicking on it. We now take a look at the synthesized source code to see if the RTL refinement tool has correctly generated the RTL model. Do this by selecting View⟶Source from the menu bar.

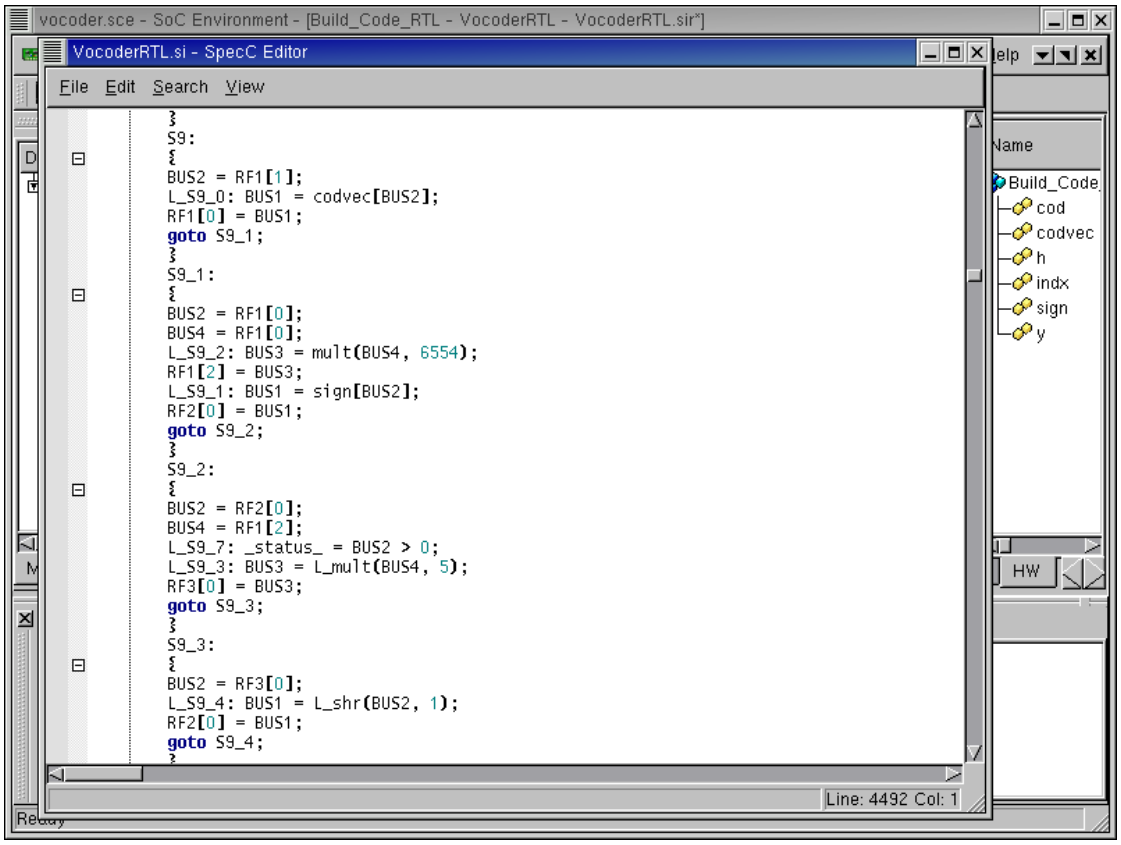
If the reader is not interested, he or she may skip this section to go directly to Section 4.5.4 *View Verilog RTL model (optional)* (page 217).

4.5.3.1. View RTL model (optional) (cont'd)



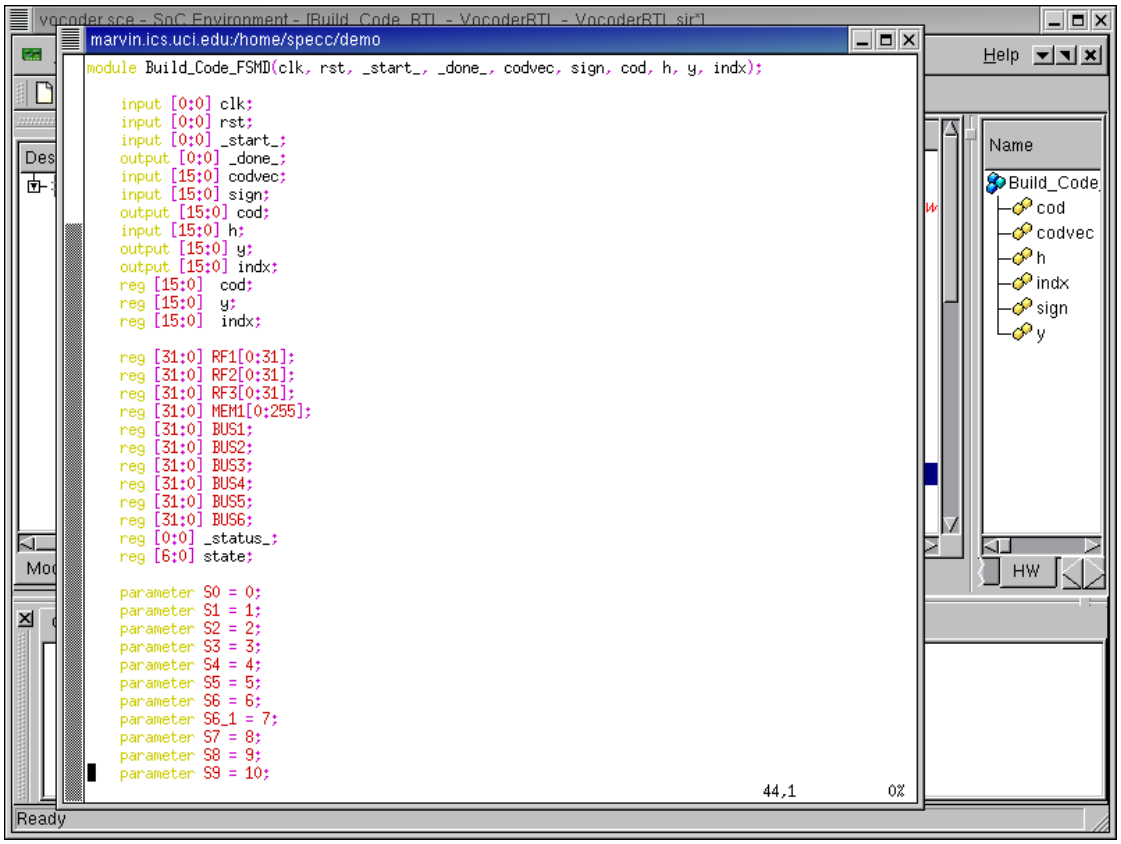
The SpecC Editor pops up showing the RTL code for behavior, "Build_Code_RTL." Scrolling down the editor window shows several function declarations in this behavior. It is to be noted that these declarations correspond to the functions implemented for the allocated RTL components. Also, we can observe a FSMD construct with 10 ns clock period.

4.5.3.2. View RTL model (optional) (cont'd)



Scrolling down further shows the assignments for the state variables. Recall that the RTL synthesis produced 112 states. These states are enumerated here from 0 through 111. Note the final assignment (S_EXIT = 111). Further observations of the generated code show read/write operations on the register files. For instance, RF1 is the register file written in the statement RF1[0] = BUS1; as shown in state S9.

4.5.4. View Verilog RTL model (optional)

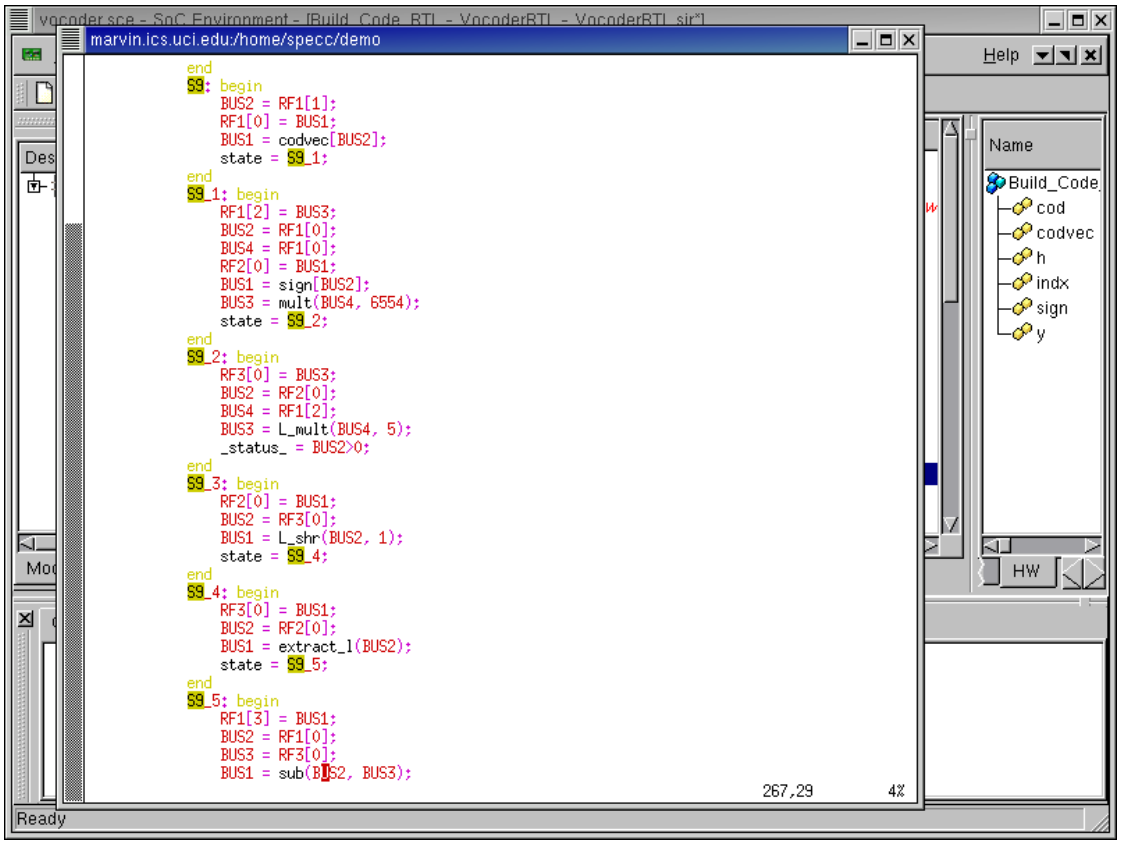


Check out the Verilog code generated in the file VocoderRTL.v. This code is generated by the RTL refinement tool. The designer may go the shell and launch his favorite editor to browse through the generated Verilog code.

If the reader is not interested, she or he can skip this section to go directly to Section 4.5.5 *Simulate RTL model (optional)* (page 219).

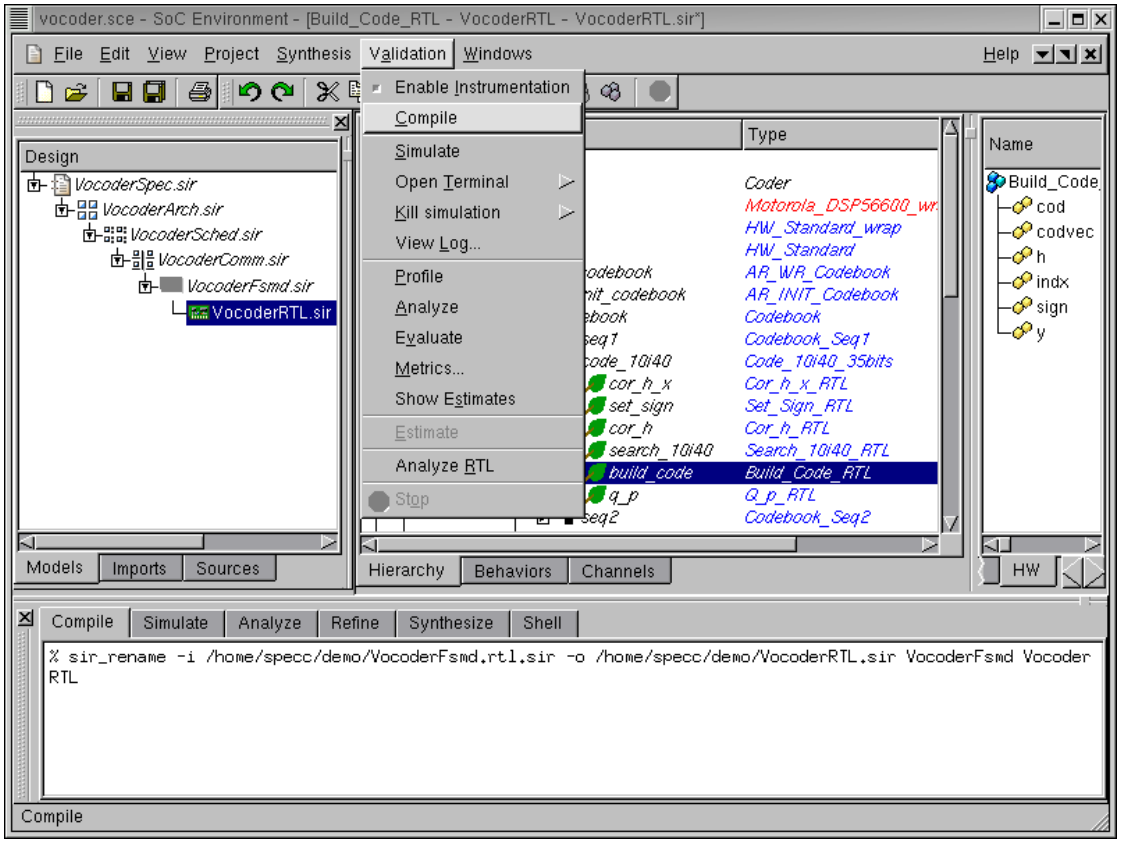
Note that the Verilog code has corresponding modules for 6 sub-behaviors of Code_10i40_35bits.

4.5.4.1. View Verilog RTL model (optional) (cont'd)



In the Verilog code, we use "case" construct to represent FSMD. All states are defined by parameter construct. If "_start_" signal is activated, FSMD begins to execute and then if FSMD reaches state S_EXIT, "_done_" signal is asserted and FSMD will end to execute and will wait for the next entry of execution.

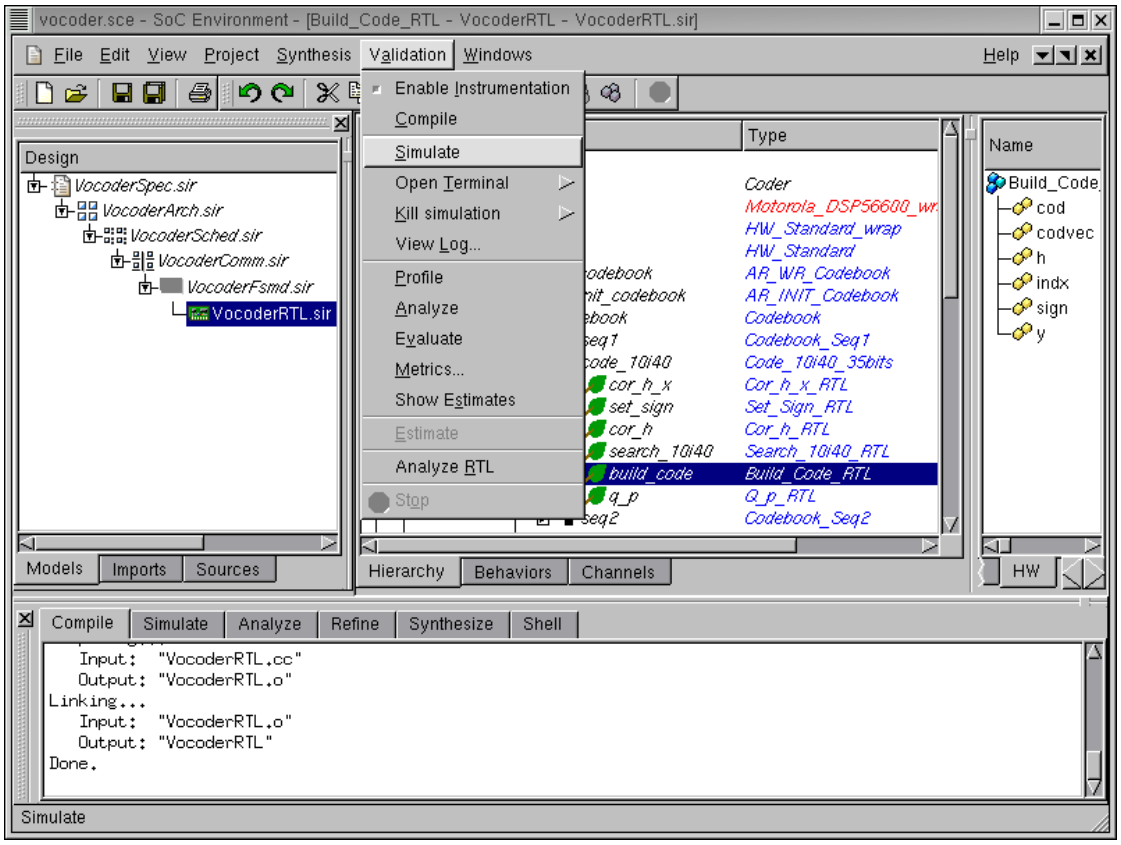
4.5.5. Simulate RTL model (optional)



Now, we have to create an executable for the generated FSMD model by selecting Validation⟶Compile from the menu bar.

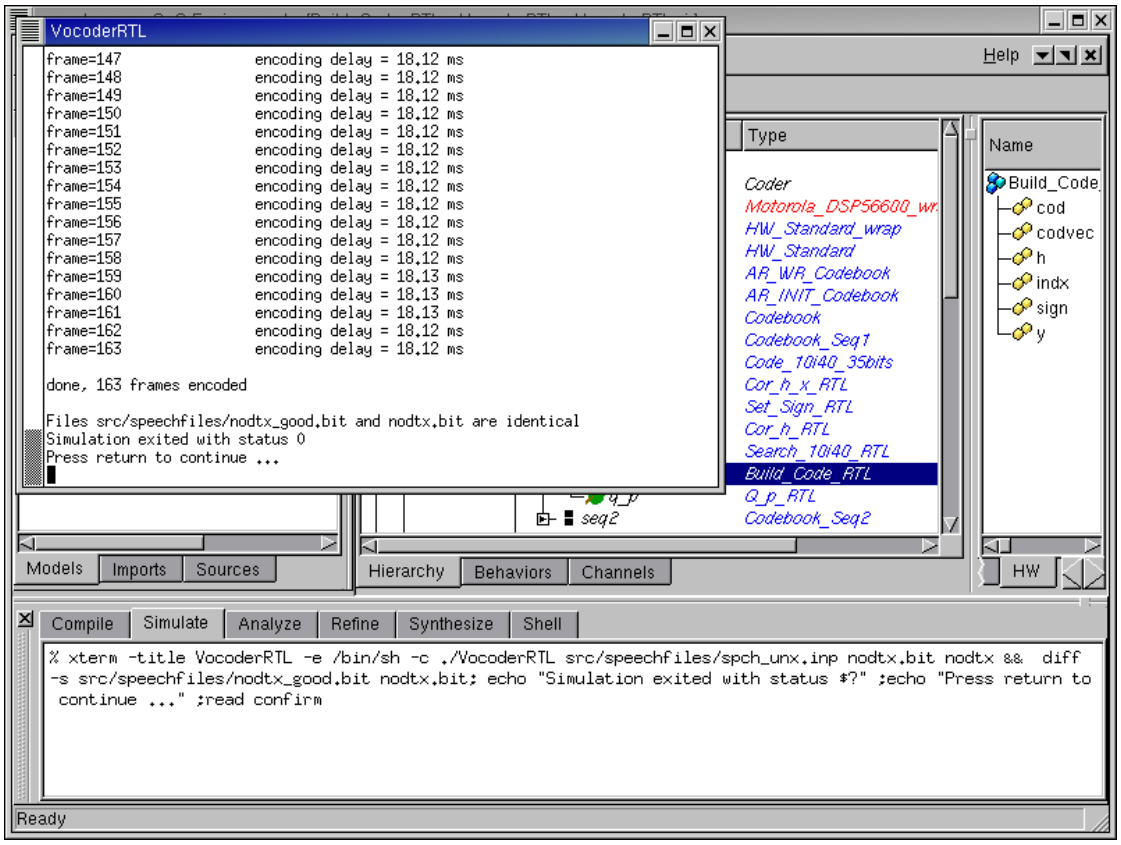
If reader is not interested, she or he can skip this section to go directly Chapter 5 *Embedded Software Design* (page 223).

4.5.5.1. Simulate RTL model (optional) (cont'd)



Note that the RTL model compiles correctly generating the executable VocoderRTL as seen in the logging window. We now proceed to simulate the model by selecting Validation⟶Simulate from the menu bar.

4.5.5.2. Simulate RTL model (optional) (cont'd)



The simulation window pops up showing the progress and successful completion of simulation. We are thus ensured that the RTL refinement step has taken place correctly. Also note that we can perform the RTL refinement on any behavior of our choice. This indicates that the user has complete freedom of delving into one behavior at a time and testing it thoroughly. Since the other behaviors are at a higher level of abstraction, the simulation speed is much faster than the situation when the entire model is synthesized. This is a big advantage with our methodology and it enables partial simulation of the design. The designer does not have to refine the entire design to simulate just one behavior in RTL.

In this simulation, we see the delay per frame in RTL model increases to 18.13 ns from 17.05 ns compared to SFSMD model. Because each state in the SFSMD model is split into multiple states by scheduling and binding.

4.6. Summary

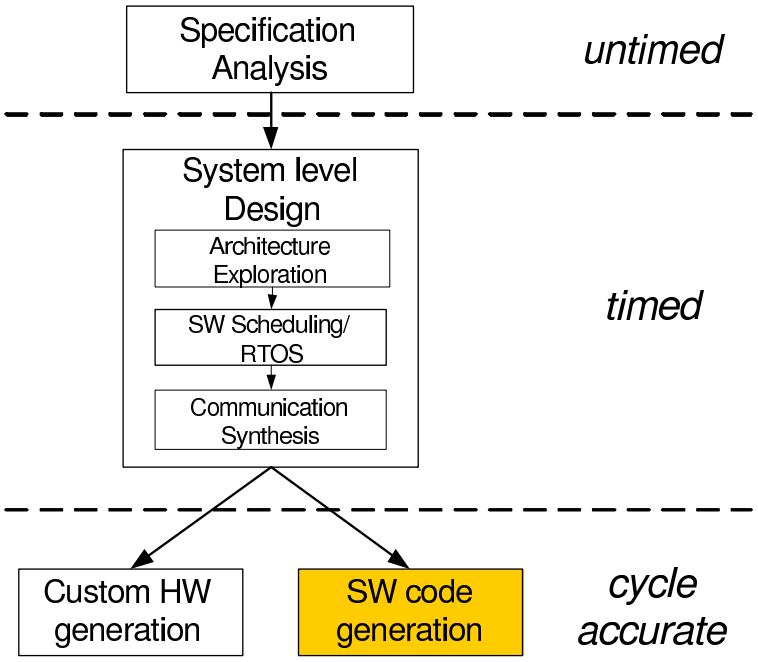
In this chapter we showed the task of custom HW design for the behaviors mapped to HW component. We started from a bus functional model of the system and isolated the behaviors that we want to implement in HW. These behaviors underwent a series of transformations to arrive at a FSMD style model that can serve as input to industry standard logic synthesis tools. Besides, generating the SpecC models, SCE is also capable of generating HW models in standard HDL like Verilog and Handel-C, which can be used by the Celoxica Design Kit.

We also saw various advantages of working with SCE during RTL synthesis. The environment and language allow the user to concentrate only on one behavior if he or she needs to. That is, the designer may choose to perform cycle a accurate implementation of a critical behavior and keep the remaining behaviors at a higher level of abstraction for fast simulation. The RTL synthesis process itself allows the designer to perform the scheduling and binding steps manually. However, we also showed the automatic RTL synthesis capabilities. The designer is free to tweak the synthesis results and generate a new model at any time.

Chapter 5. Embedded Software Design

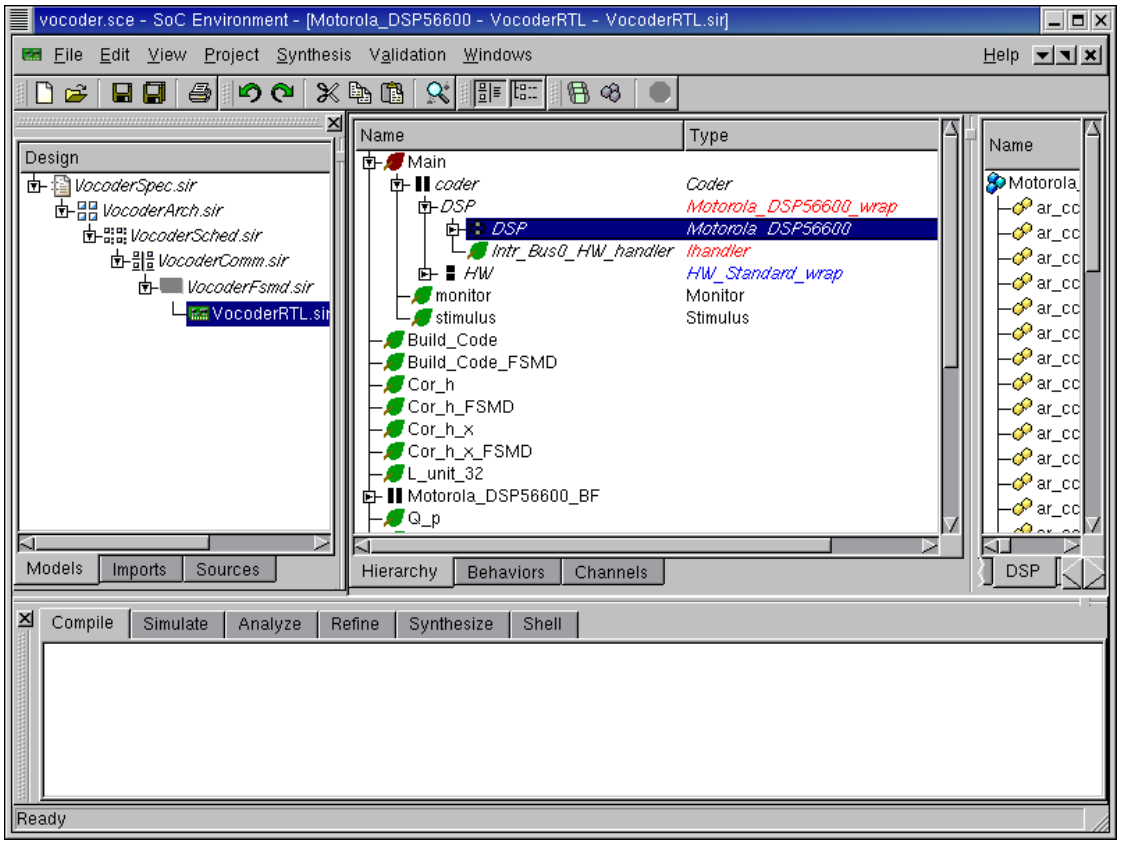
5.1. Overview

Figure 5-1. SW code generation with SCE



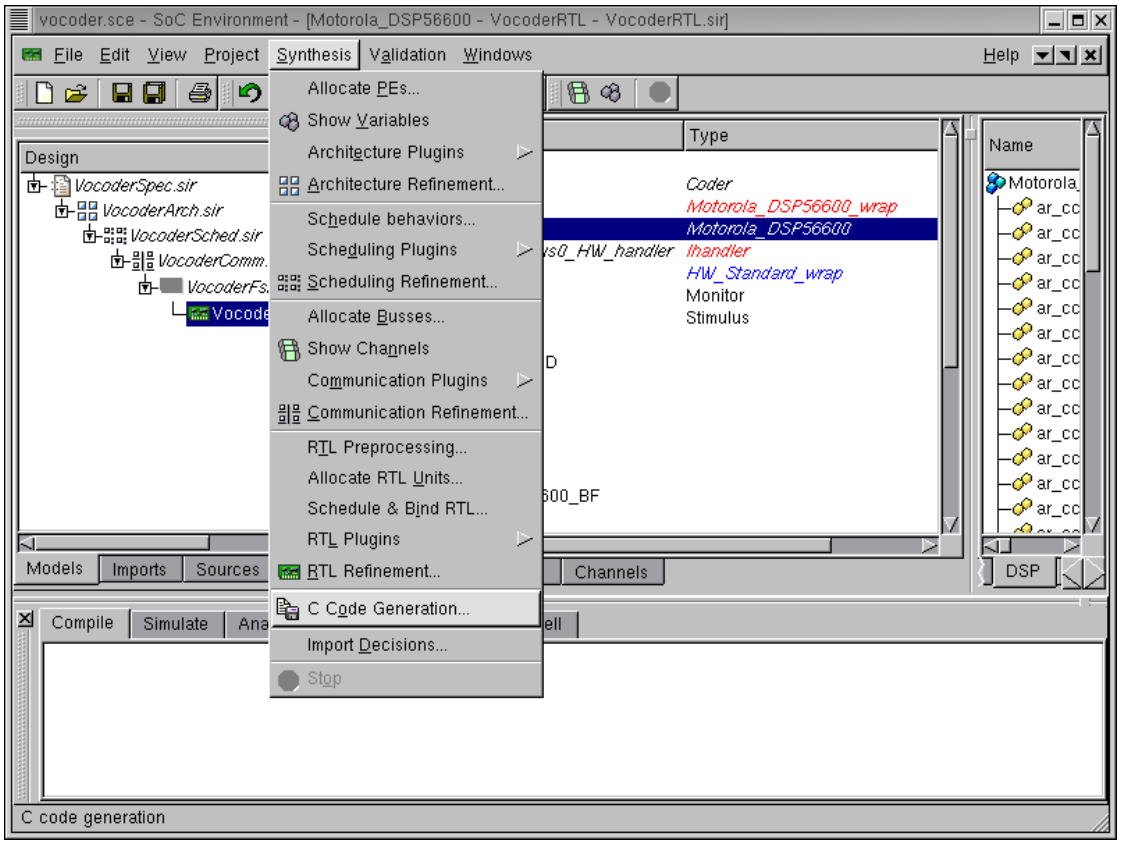
In this chapter, we look at software code generation as highlighted in Figure 5-1. The bus functional model derived after system level design contains a behavioral hierarchy of tasks mapped to SW components. Since the SpecC code is not a natural input for generating the processor's instruction-set specific code, we need to produce C code that can be compiled for the processor. In this phase we use the SW generation tool to flatten the hierarchical SpecC code and produce C code. We thus enable the designer to use an off the shelf processor with C compiler and produce cycle accurate SW for it. The instruction set simulator for the processor can be used in conjunction with the SpecC simulator to perform cycle accurate simulation of both HW and SW.

5.2. SW code generation



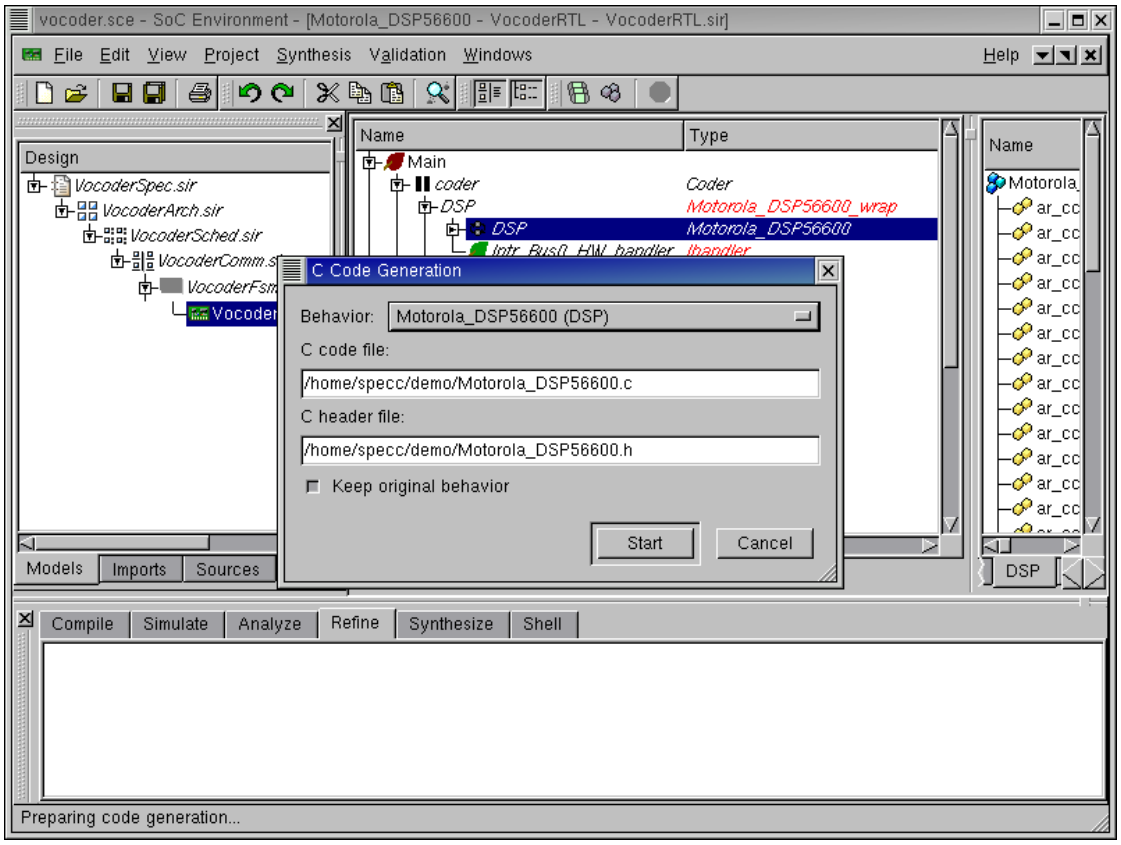
Once we are done with HW and have obtained a RTL model, we will generate software for the DSP. For our design example, we need to generate C code for behavior "Motorola_DSP56600" and all its child behaviors. We start by selecting behavior "Motorola_DSP56600" in the design hierarchy tree.

5.2.1. Generate C code



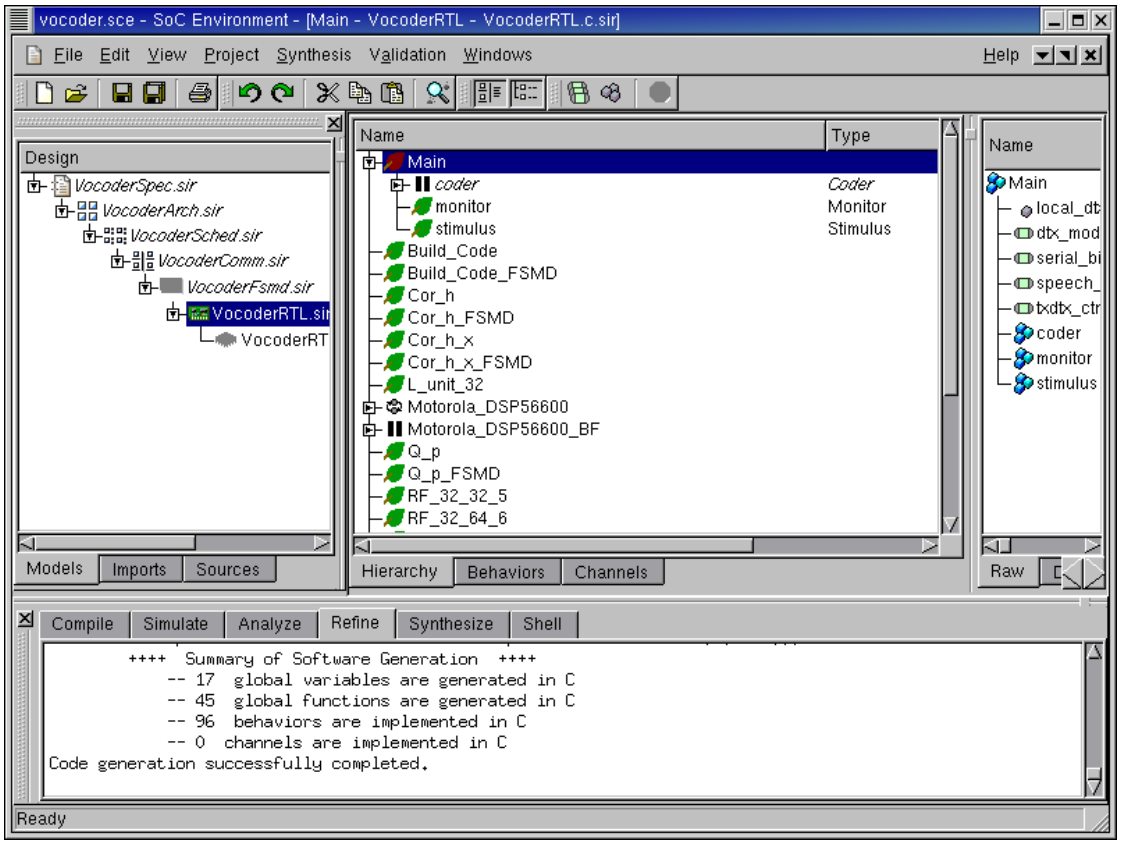
To generate C code for behavior "Motorola_DSP56600", select Synthesis⟶C Code Generation... from the menu bar.

5.2.1.1. Generate C code (cont'd)



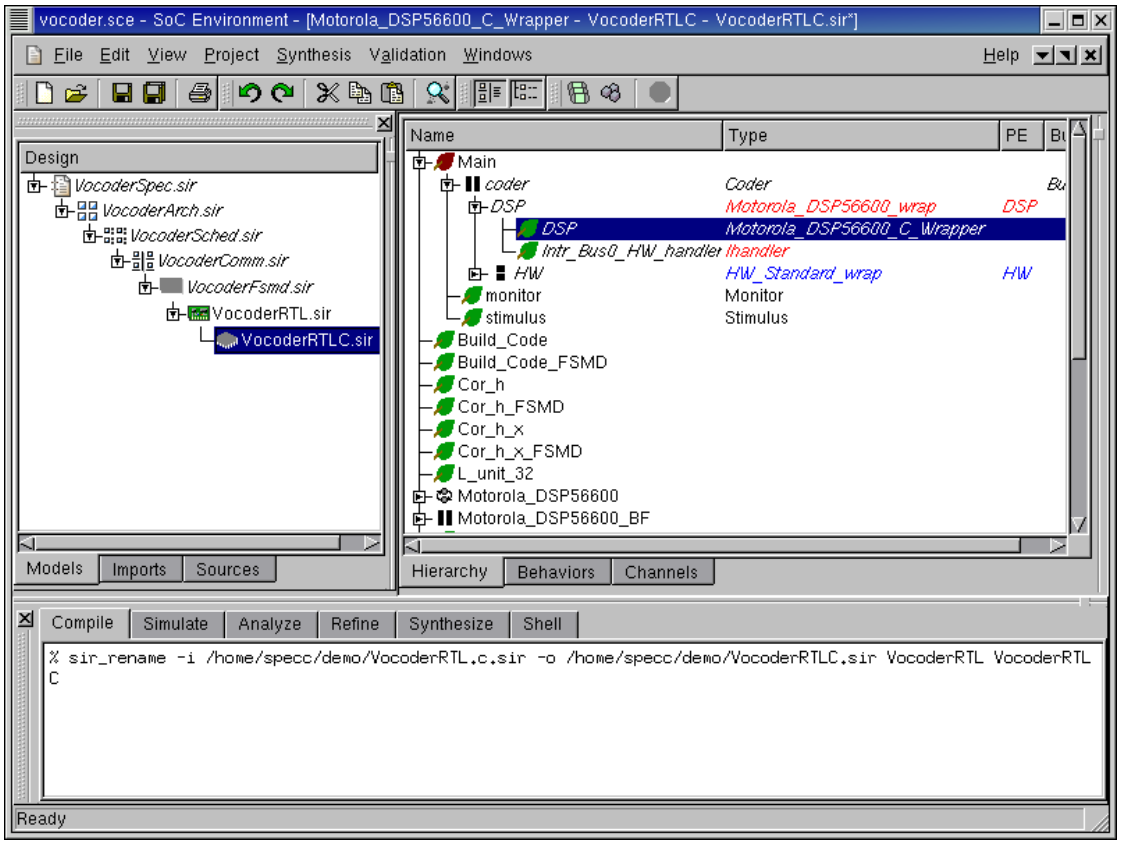
A dialog box pops up for the user to input the name of the C and Header file of the generated software. Now press the Start button to start the C code generation process.

5.2.1.2. Generate C code (cont'd)



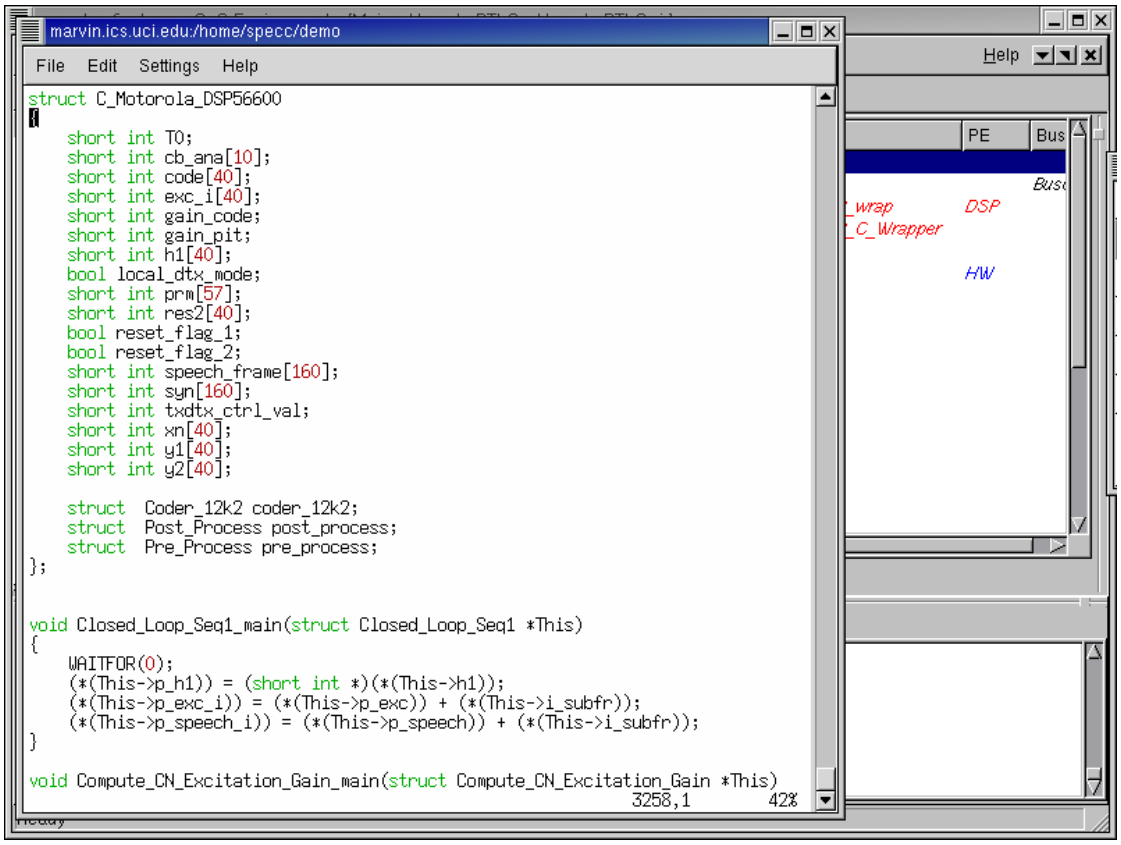
As displayed in the logging window, the software generation is being performed. The newly generated software model "VocoderRTL.C.sir" is displayed to the design window. It is also added to the current project window, under the RTL model "VocoderRTL.sir" to indicate that it was derived from "VocoderRTL.sir"

5.2.1.3. Generate C code (cont'd)



Like in the previous sections, we need to change the design name to follow the same naming style in this tutorial. In the project window, select design "VocoderRTL.C.sir". Right click and select Rename... Change the design name to "VocoderRTLC.sir"

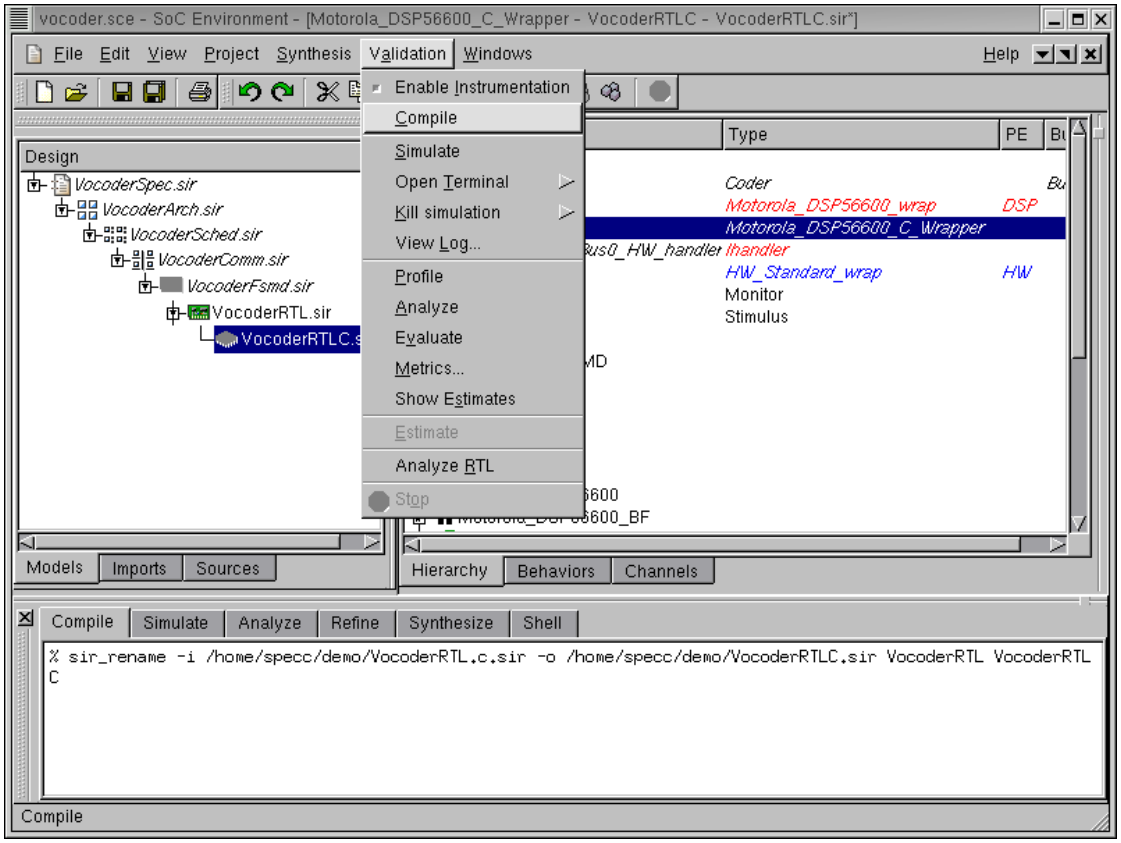
5.2.2. Browse and View C code



Check out the C code generated in the file "Motorola_DSP56600.c". This code is generated by the software generation tool. The designer may go to the shell and launch his favorite editor to browse through the generated C code.

The code generation process converts the SpecC description of tasks into ANSI C code. The main idea is that we convert the behaviors and channels into C struct and convert the behavioral hierarchy into the C struct hierarchy. Variables defined inside a behavior or channel and ports of behaviors are converted into data members of the corresponding C struct. Finally, functions inside a behavior or channel are converted into global functions with an additional parameter added representing the behavior to which the function belongs.

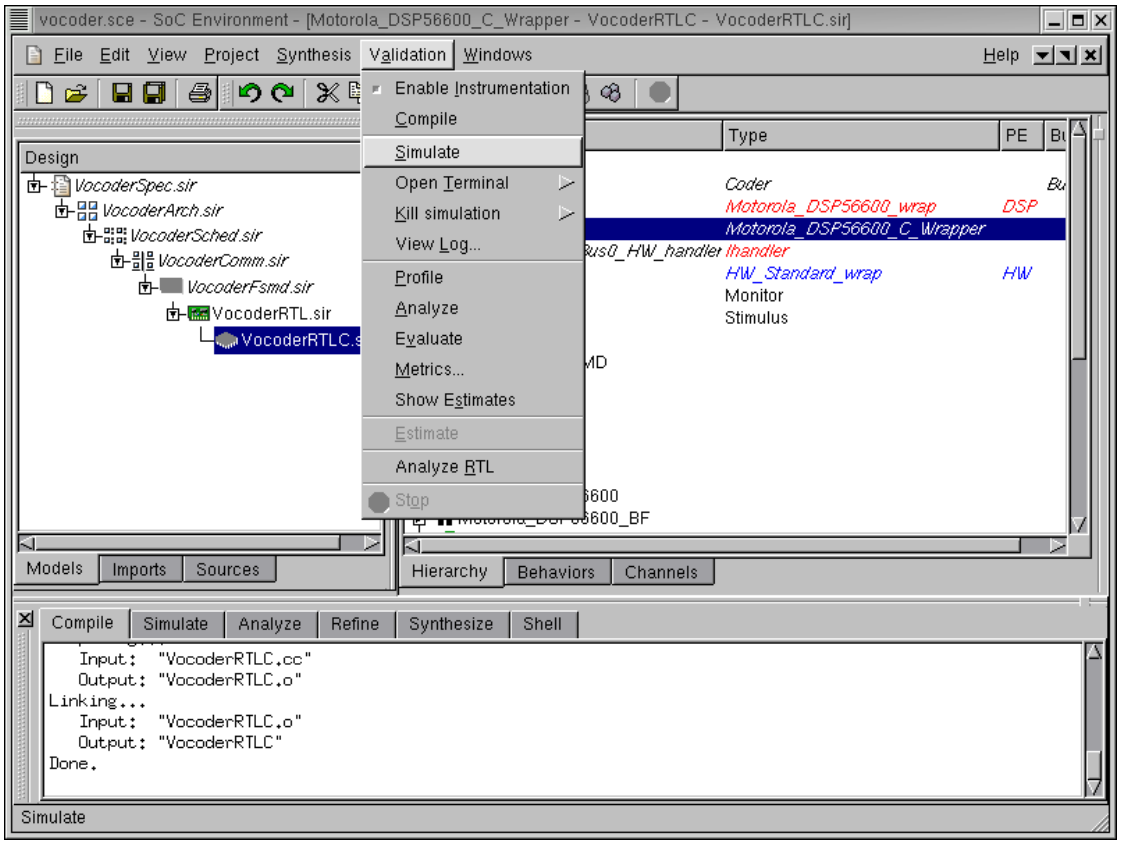
5.2.3. Simulate C model (optional)



So far we have finished the C code generation. However, we also need to confirm that the generated C code is correct for the design. In other words the C code must be functionally equivalent to the SpecC model. The simulation step is optional, so if the designer is not interested in it, he or she may skip it and go directly to Section 5.3 *Instruction set simulation* (page 233).

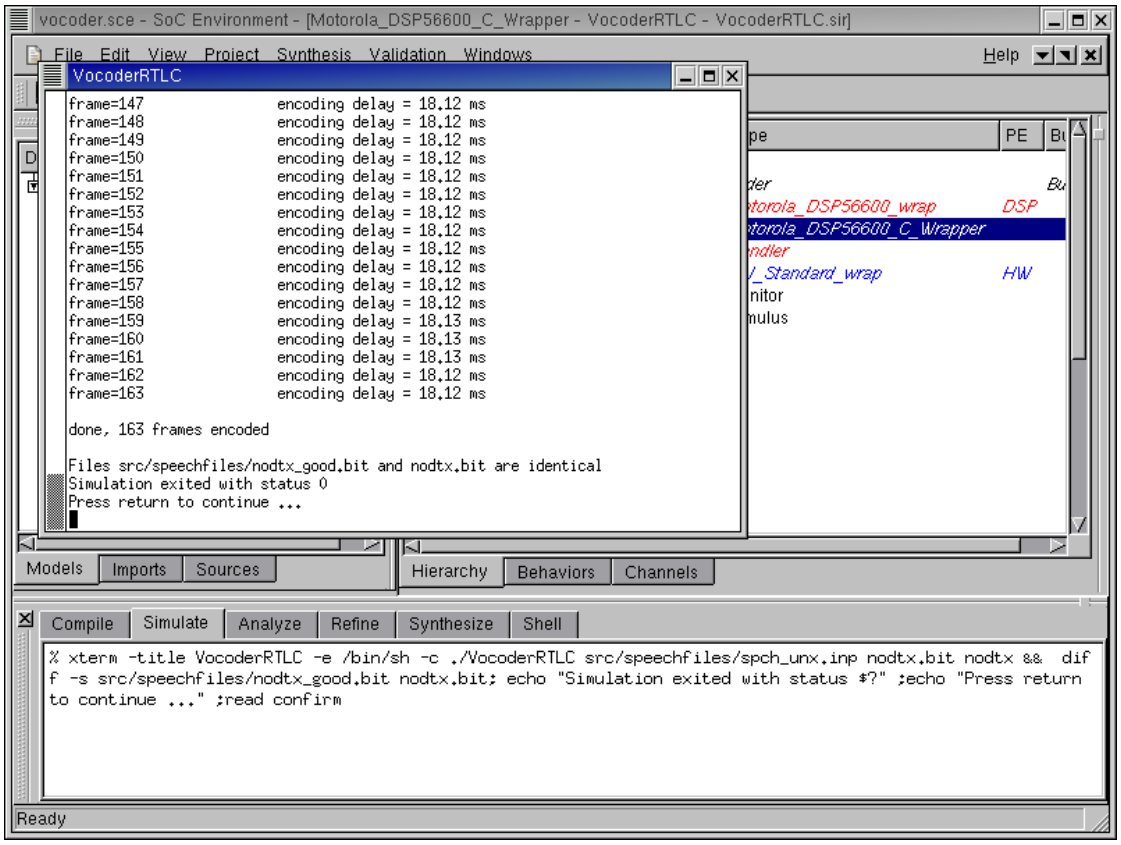
We will validate the generated C code through simulation. But first we need to import C code into the design and compile the model into an executable. To compile the C code model to executable, go to Validation menu and select Compile .

5.2.3.1. Simulate C model (cont'd)



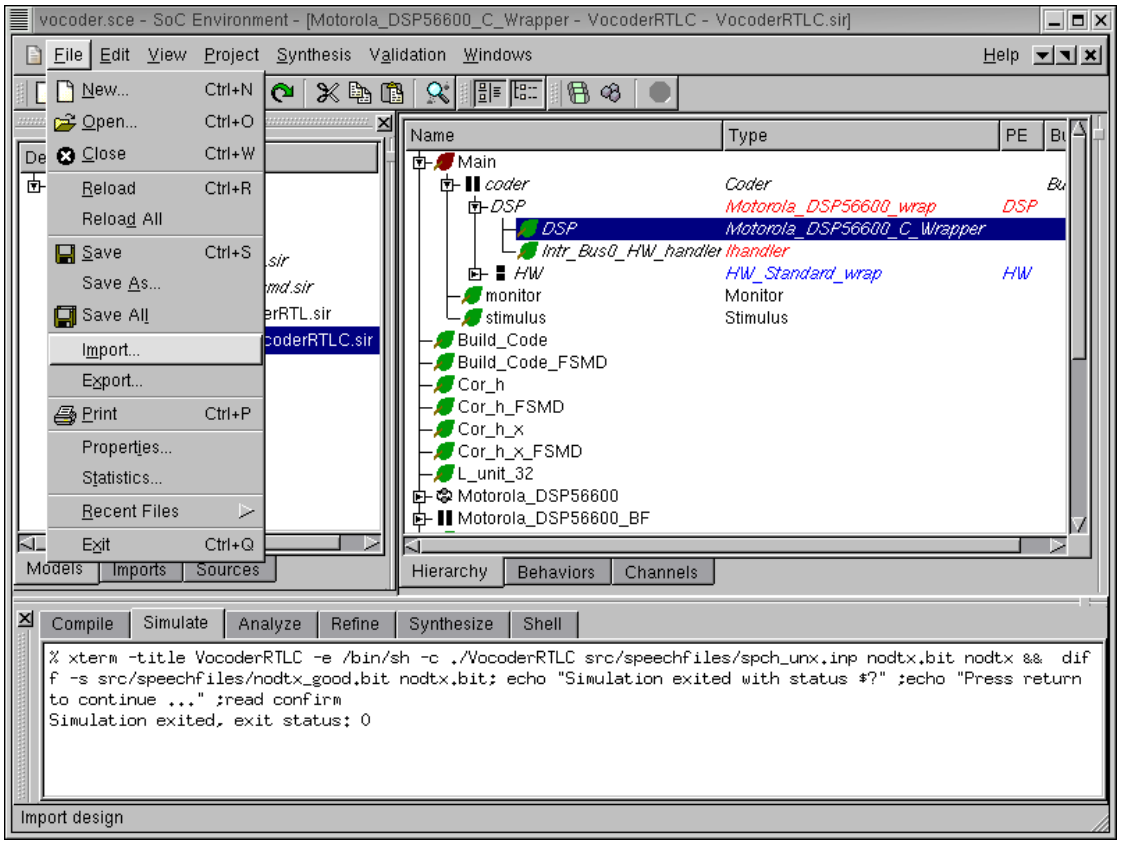
The messages in the logging window shows that the C code model is compiled successfully without any syntax errors. Now in order to verify that it is functionally equivalent to the previous model, we will simulate the compiled model on the same set of speech data used in the specification validation. Go to Validation menu and select Simulate .

5.2.3.2. Simulate C model (cont'd)



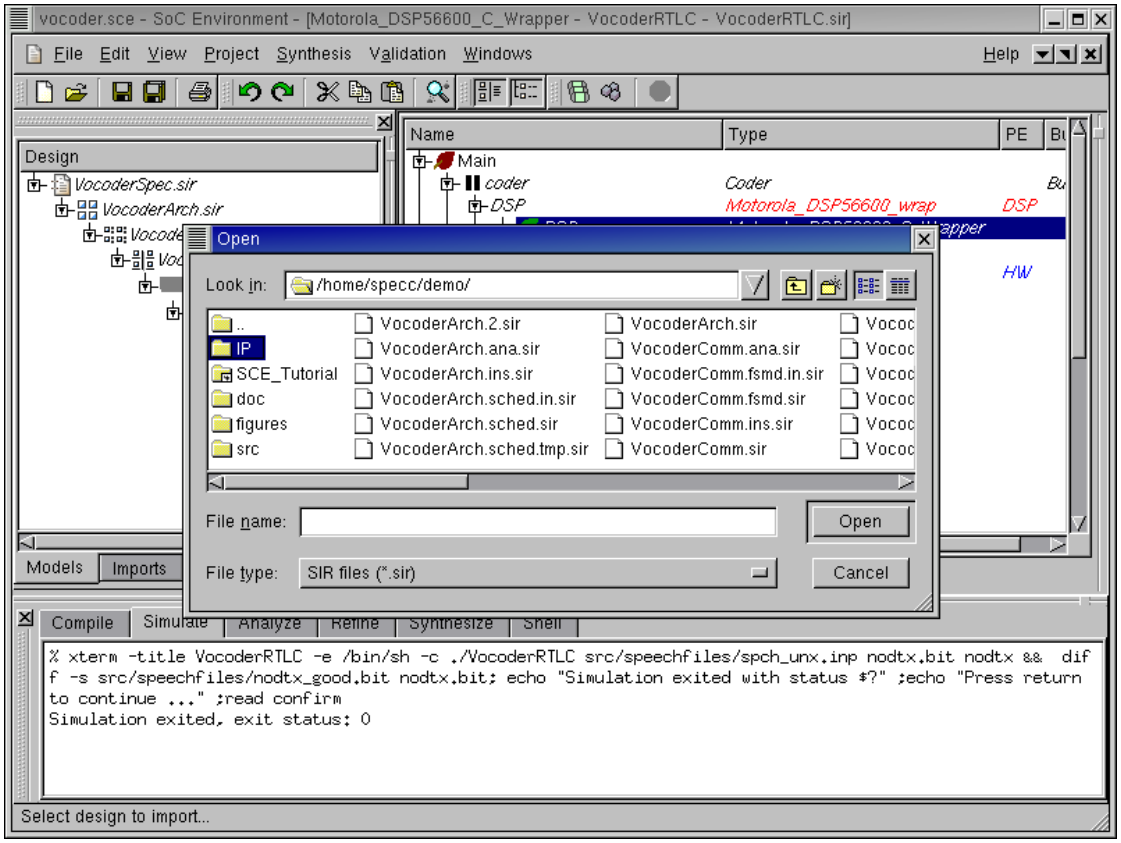
Like in the earlier cases, a simulation window pops up. The simulation result is correct and we have thus verified that the generated C code is functionally correct.

5.3. Instruction set simulation



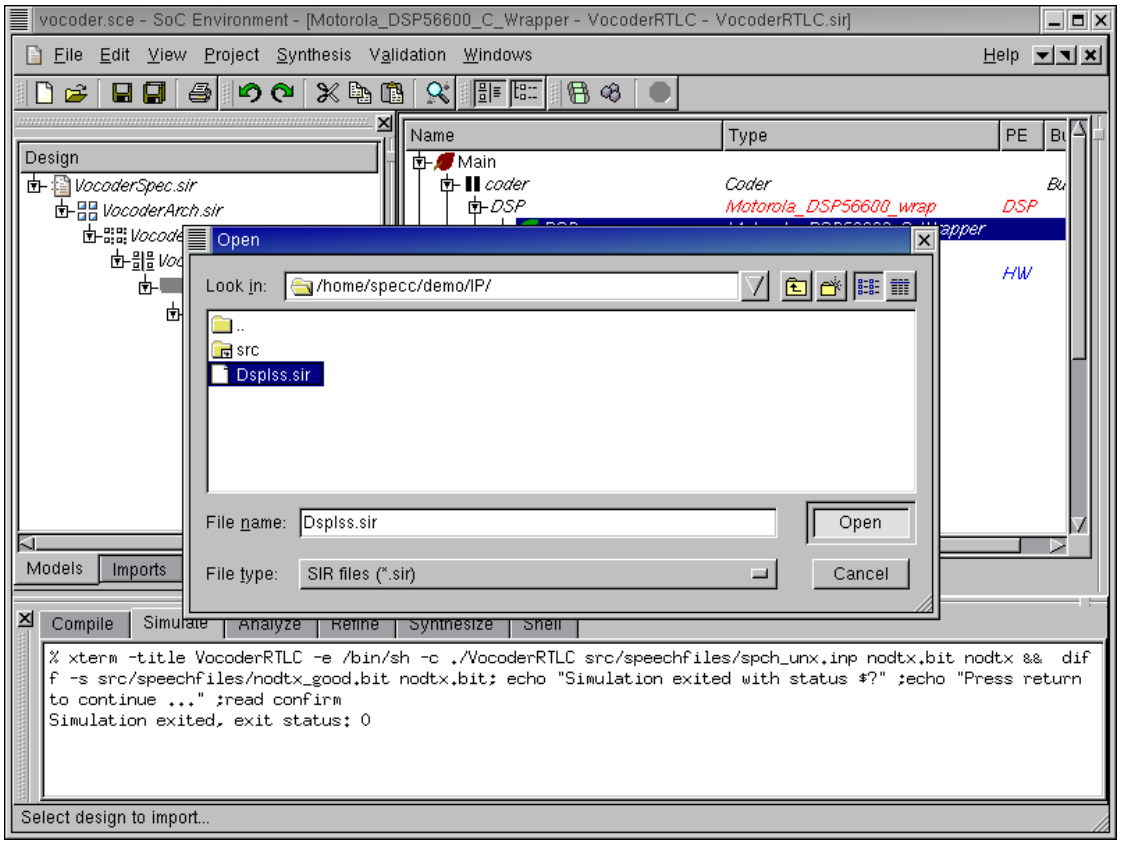
After we generated C code for the DSP, we compile the C code into DSP's instruction set and import the instruction set simulator (ISS) for the Motorola DSP56600. To start importing, select File⟶Import from the menu bar.

5.3.1. Import instruction set simulator model



Select directory "IP" from the file selection menu by double Left click.

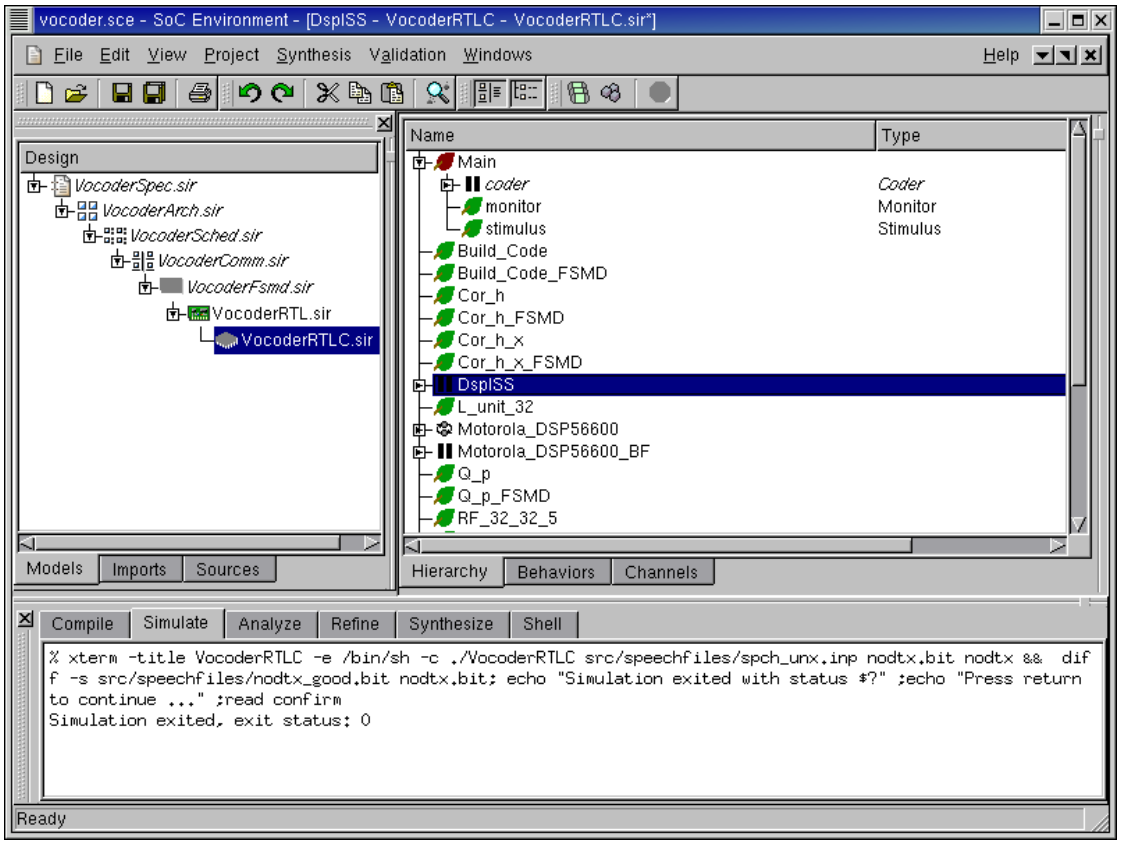
5.3.1.1. Import instruction set simulator model (cont'd)



Inside directory IP, select "DspIss.sir" and Left click on Open.

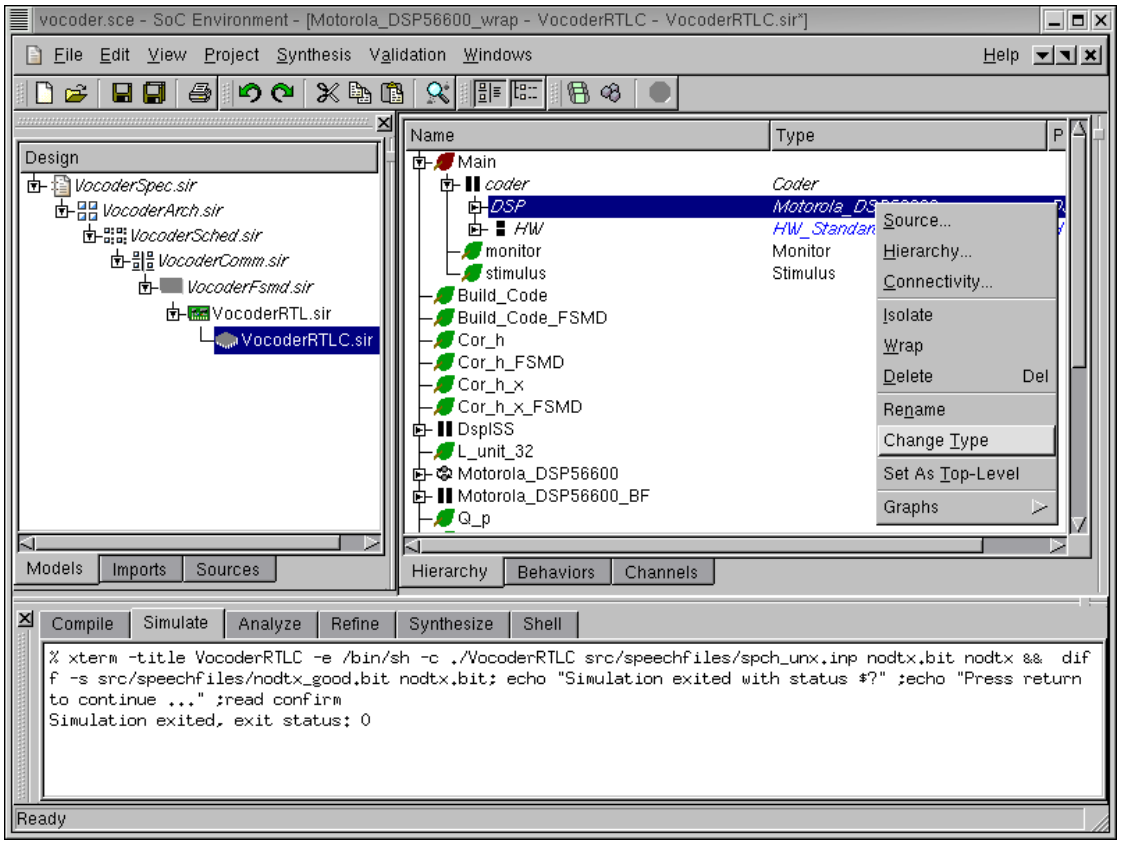
The SIR file contains the instruction set simulator for our chosen DSP. The behavior loads the compiled object code for the tasks that were mapped to DSP and executes it on the instruction set simulator.

5.3.1.2. Import instruction set simulator model (cont'd)



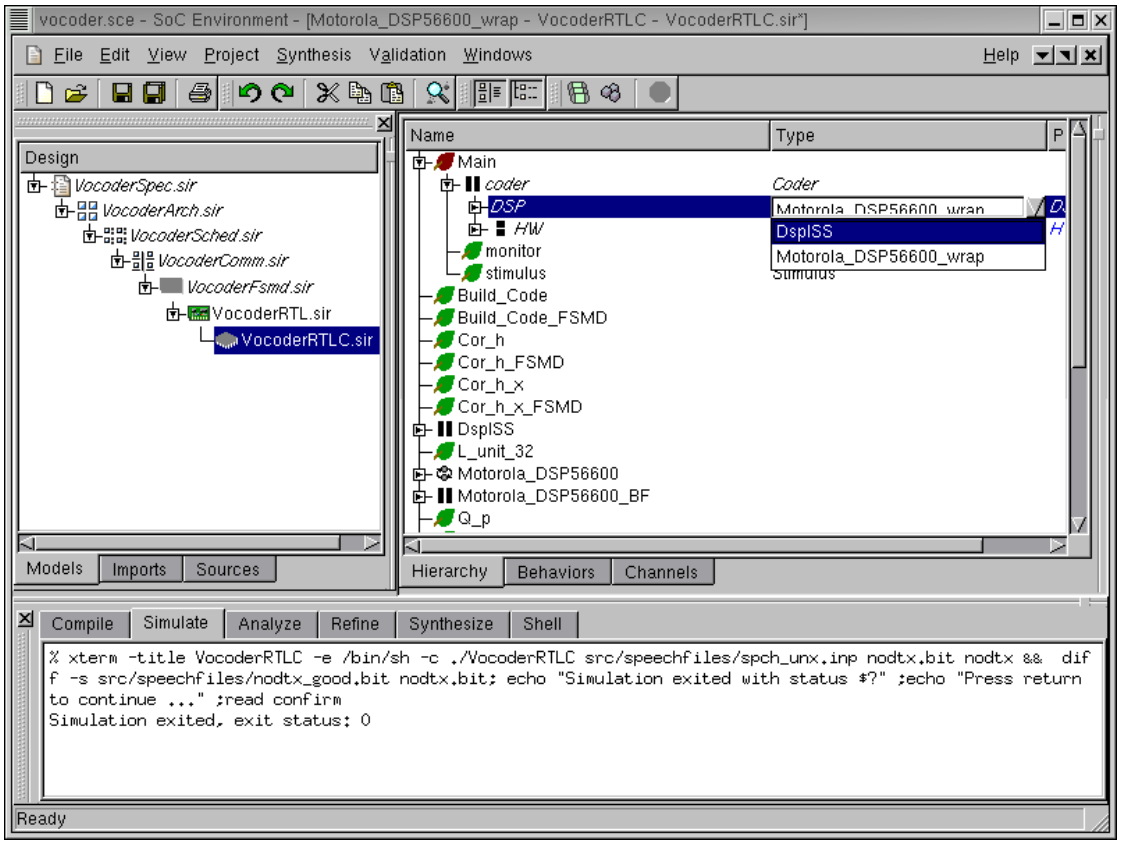
Once "DspIss.sir" is imported, we can notice behavior "DspISS" as a new root behavior in the design hierarchy tree. This is because behavior "DspISS" has not been instantiated yet.

5.3.1.3. Import instruction set simulator model (cont'd)



In the design hierarchy tree, select behavior "DSP". Right click and select Change Type.

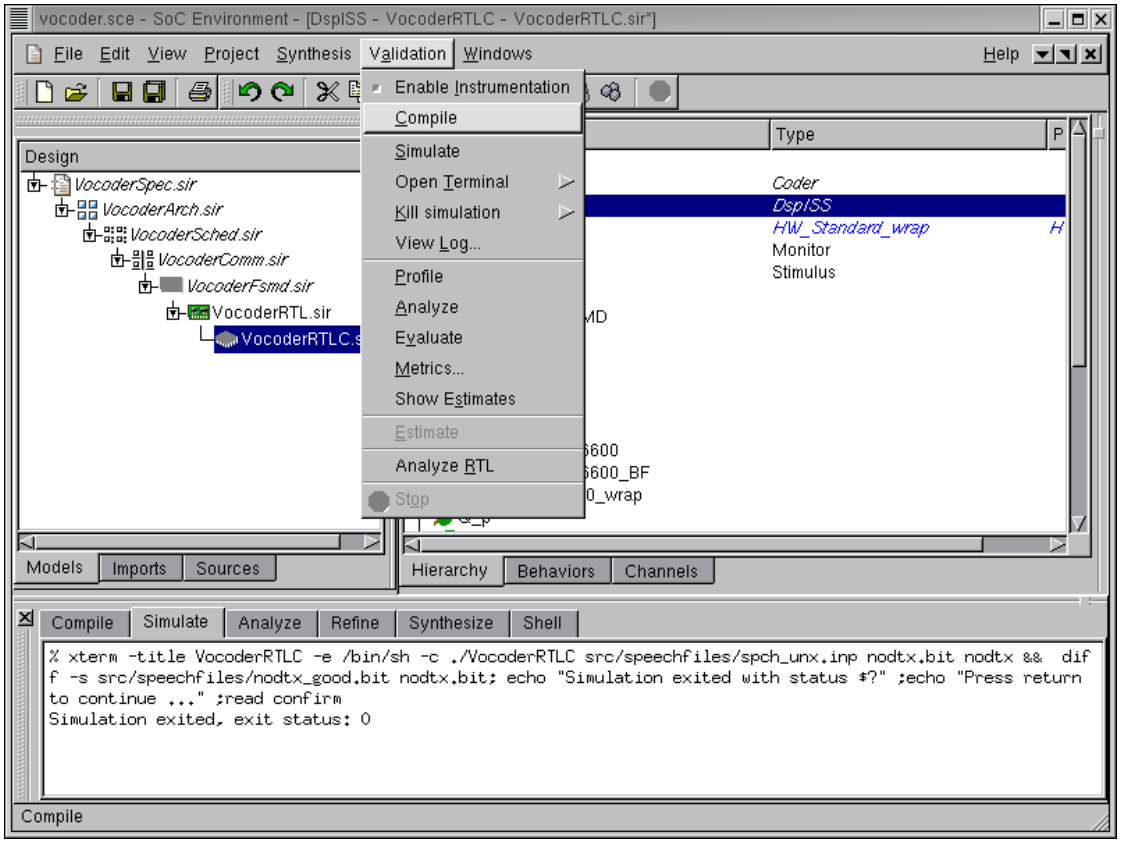
5.3.1.4. Import instruction set simulator model (cont'd)



The type of behavior "DSP" may now be changed by selecting DspISS.

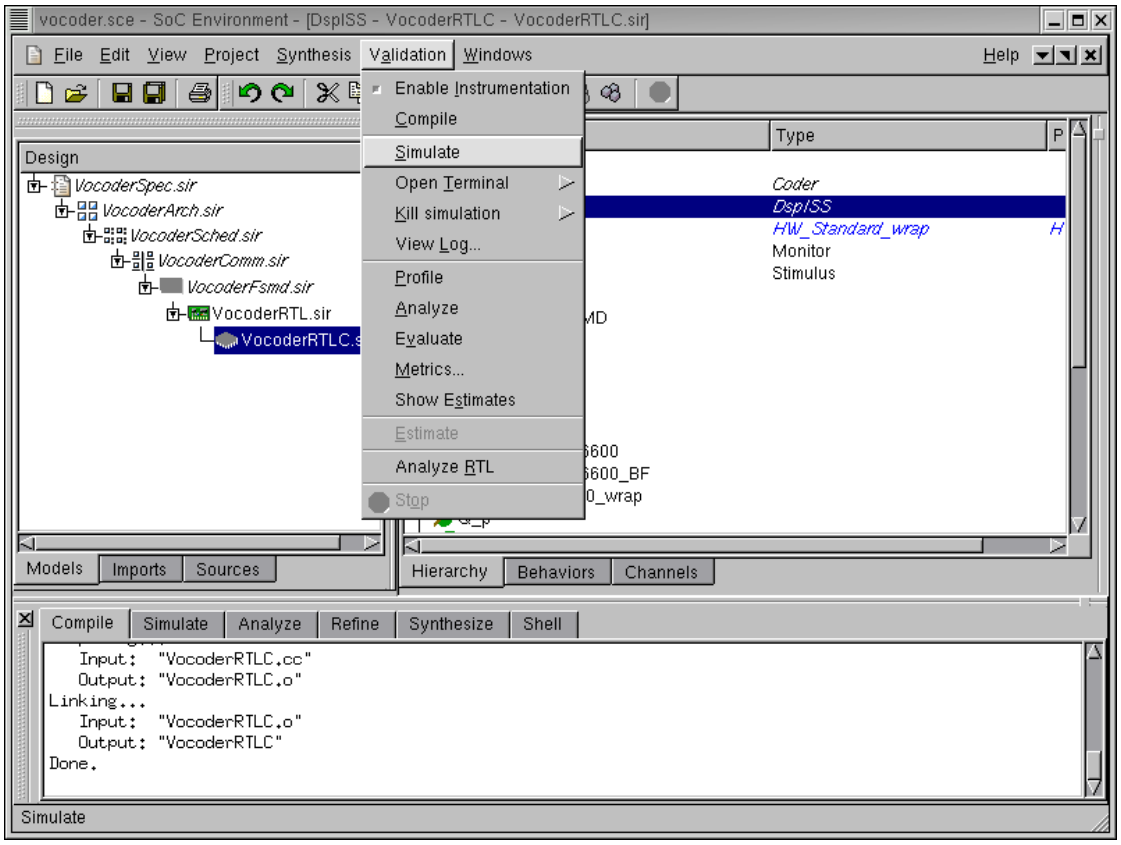
By doing this, we have now refined the software part of our design to be implemented with the DSP56600 processor's instruction set. Recall that the software part mapped to DSP has already been compiled for the DSP56600 processor and the object file is ready. As mentioned earlier, the new behavior will load this object file and execute it on the DSP's instruction set simulator. Thus the model becomes clock cycle accurate.

5.3.2. Simulate cycle accurate model



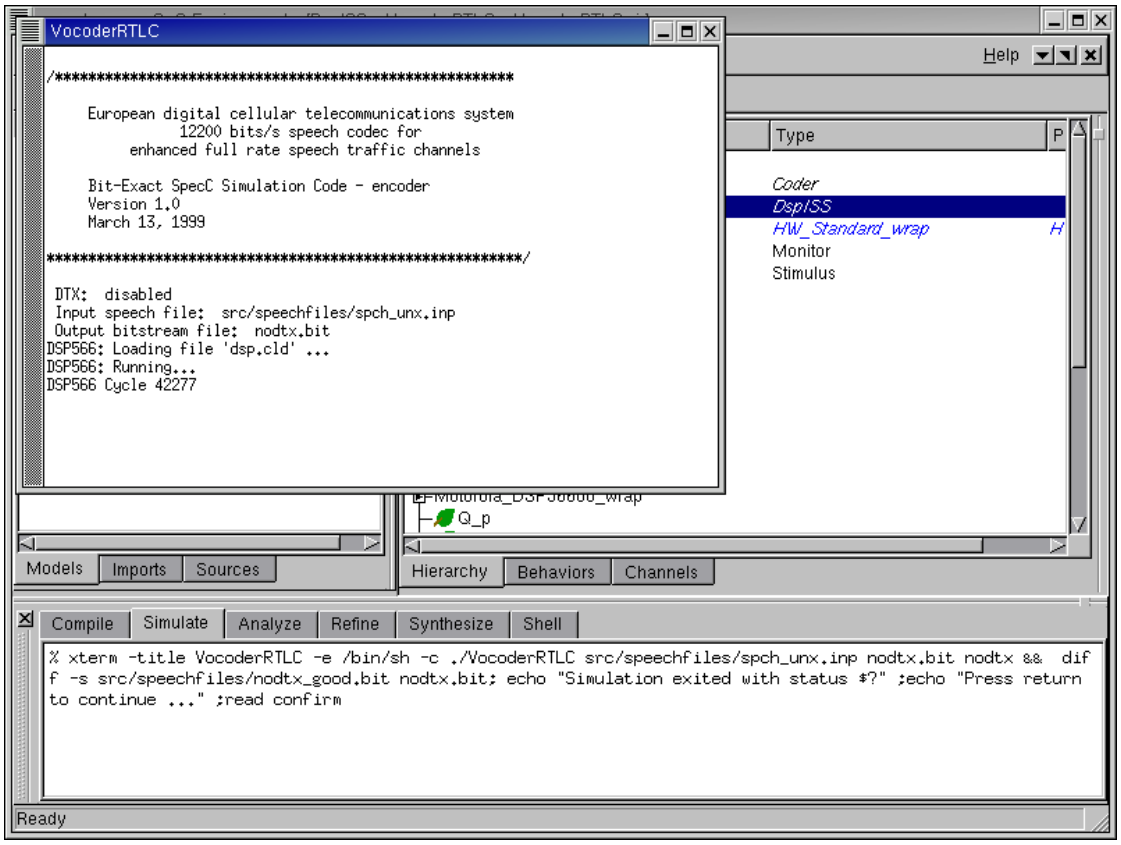
We now have the clock cycle accurate model ready for validation. We begin as usual with compiling the model by selecting Validation⟶Compile from the menu bar.

5.3.2.1. Simulate cycle accurate model (cont'd)



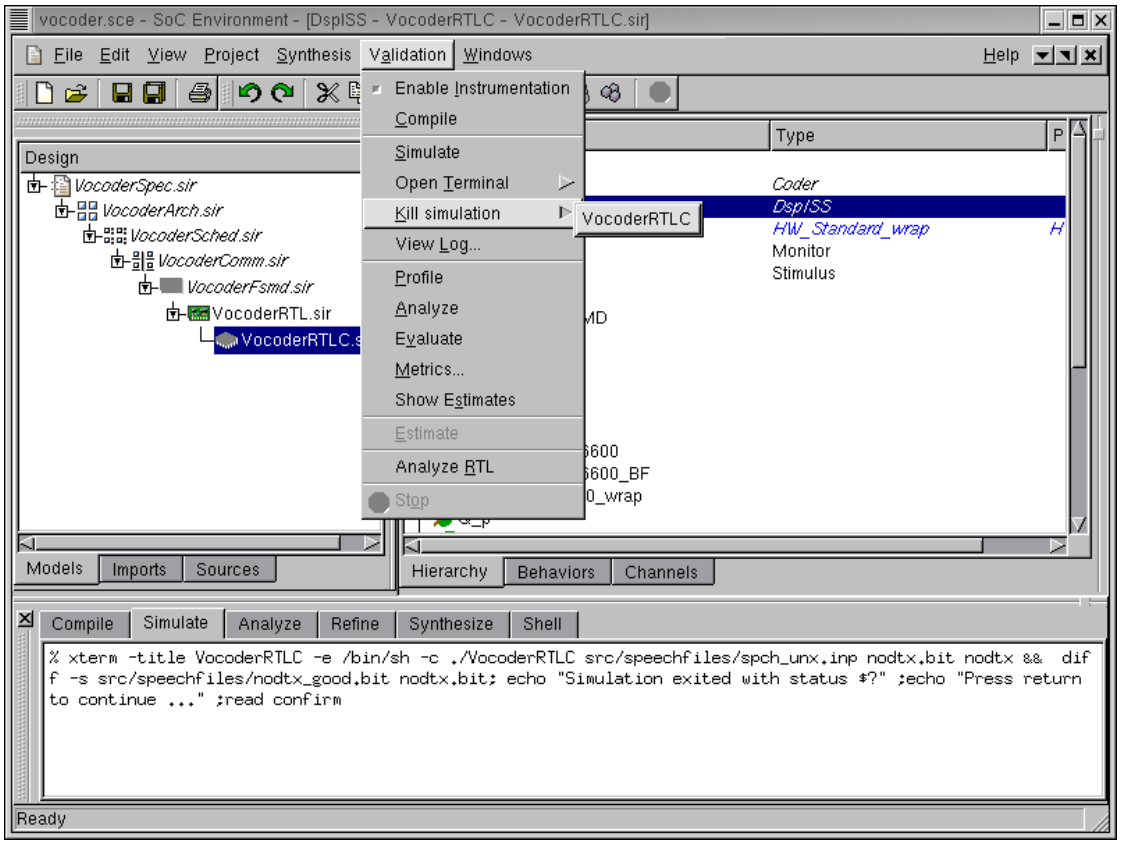
The model compiles correctly as shown in the logging window. We now proceed to simulate the model by selecting Validation⟶Simulate from the menu bar.

5.3.2.2. Simulate cycle accurate model (cont'd)



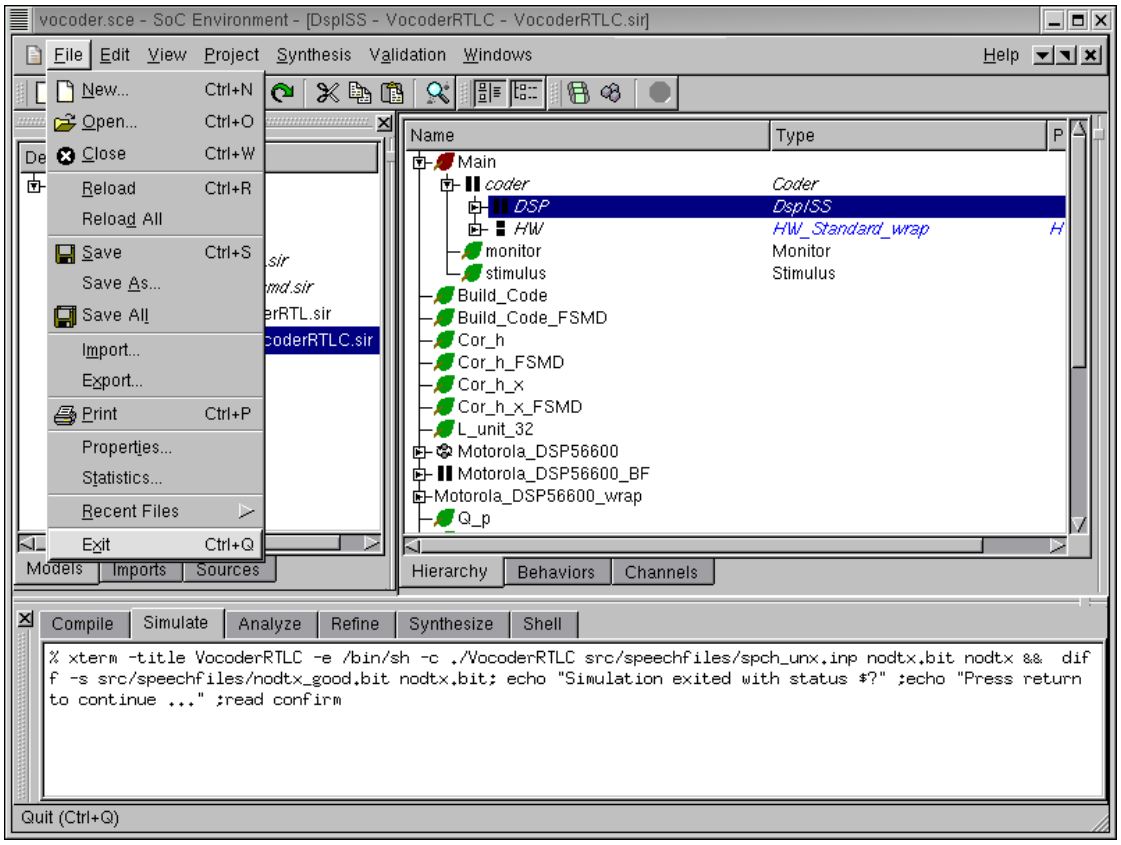
Like in the earlier cases, a simulation window pops up. The DSP Instruction set simulator can be seen to slow down the simulation speed considerably. This is because the simulation is being done one instruction at a time in contrast to the high level simulation we had earlier.

5.3.2.3. Simulate cycle accurate model (cont'd)



It may take hours for the simulation to complete. The simulation may be killed by selecting Validation⟶Kill simulation from the menu bar.

5.3.2.4. Simulate cycle accurate model (cont'd)



The demo has now concluded. To exit the SoC environment, select Project⟶Exit from the menu bar.

5.4. Summary

In this tutorial, we performed the SW synthesis task after RTL synthesis of HW. Note that these two tasks are orthogonal and may be done in any order. We showed C code generation for the behaviors mapped to SW component. This is a useful feature of SCE, since we can generate C code which can be compiled onto any processor to generate assembly. The code can then used for an instruction set simulator to run on a cycle-by-cycle basis with the RTL HW. All these built in features of SCE allow the designer to move across abstraction levels even for parts of a design. The flexibility and design capablity that is thus provided to the designer is enormous.

Chapter 6. Conclusion

In this tutorial we presented the System on Chip design methodology. The SoC methodology defines the 4 models and 3 transformations that bring an initial system specification down to an RTL-implementation. In addition to validation through simulation, the well-defined nature of the models enables automatic model refinement, and application of formal methods, for example in verification.

The complete design flow was demostrated on an industrial strength example of the Vocoder Speech encoder.We have shown how SCE can take a specification model and allow the user to interactively provide synthesis decisions. In going from specification to RTL/Instruction-set model for the GSM Vocoder, we noted that compared to traditional manual refinement, the automatic refinement process gives us more than a 1000X productivity gain in modeling, since designers do not need to rewrite models.

Table 6-1. Vocoder Refinement Effort

Refinement Step	Modified Lines	Manual Refinement	Automated Refinement
Spec -> Arch	3,275	3~4 months	~1 min.
Arch -> Comm	914	1~2 months	~0.5 min.
Comm -> RTL/IS	6,146	5~6 months	~2 min.
Total.	**10,355.**	**9~12 months.**	**~4 mins.**

To draw the conclusion, SCE enables the designer to use the following powerful advantages that have never been available before.

1. **Automatic model generation.**

   New models are generated by *Automatic Refinement* of abstract models. This means that the designer may start with a specification and simply use design decisions to automatically generate models reflecting those decisions.

2. **Eliminates SLDL learning.**

   SCE *eliminates the need for system-level design languages* to be learnt by the designer. Only the knowledge of C for creating specification is required.

3. **Enables non-experts to design.**

   This also *enables non-experts to design* systems. There is no need for the designer

to worry about design details like protocol timing diagrams, low level interfaces etc. Consequently, *software developers can design hardware* and *hardware designers can develop software*.

4. **Supports platforms.**

   SCE is great for *platform based design* . By limiting the choice of components and busses, designers may select their favorite architecture and then play around with different partitioning schema.

5. **Customized methodology.**

   SCE can also be *customized to any methodology* as per the designer's choice of components, system architecture, models and levels of abstraction.

6. **Enables IP trading.**

   SCE simplifies *IP trading* to a great extent by allowing interoperability at system level. With well defined wrappers, the designer can plug and play with suitable IPs in the design process. If an IP meets the design requirements, the designer may choose to plug that IP component in the design and not worry about synthesizing or validating that part of the design.

Appendix A. Frequently Asked Questions

1. **What is SCE ?**

   SCE is an acronym for System-on-Chip Environement. It is a design environement based on a model refinement methodology. The environment consists of several tools and user interfaces to help the designer take a functional system specification to its cycle accurate implementation with minimal effort.

2. **What are the supported platforms for SCE ?**

   SCE 2.2.0 beta is currently supported on Linux RedHat 7.3. The public distribution of the operating system is included on the CD-ROM. SCE has also been tested for RedHat 8.0 and SuSE 8.2 distributions of Linux. Other platforms will be supported in the future as the need arises.

3. **What is the level of expertise needed to design with SCE ?**

   SCE is designed with the goal of allowing even non-experts to perform system design. A very basic knowledge of SW and HW design, equivalent to an undergraduate degree in computer engineering, is required to work with SCE.

4. **What is the difference between behavior and model ?**

   A model is a description of the design in a machine readable form (like SpecC). There may be several models used in a system design effort. These models capture the design with varying levels of abstraction. A behavior, in context of SCE, is a unit of computation. A model is made up by a hierarchy of behaviors that communicate with each other using variables or channels.

5. **What are the models that I need ?**

   In SCE, the designer may start with only a specification model. This model captures the functionality of the design without any implementation details. As we go through the design process, various models with greater implementation details are generated automatically using the built in tools in SCE. The designer only needs to guide the model generation with decisions. The four primary models in the SCE methodology are Specification model, Architecture model, Communication model and Cycle-accurate model. The designer may choose to start with any model as per his or her choice.

6. **What do I need to do with all these models ?**

Each of the models need to be compiled to generate an executable. Once they are compiled, they need to be simulated to make sure that they work correctly. The designer may choose to view the models in graphical form to understand and verify the implementation details added as a result of refinement. The specification model also needs to be profiled to get useful data for making architectural decisions.

7. **How do I get a cycle accurate model of my design ?**

   The designer may start with any of the system level models, namely the specification model, the architecture model or the communication model. With the help of design decisions, SCE will generate subsequently refined models of the design. The final model generated after RTL refinement and SW compilation will be a cycle accurate model of the design.

8. **Why is profiling relevant ?**

   Profiling is performed to gather useful data about the specification. It gives both a quantitative and a qualitative measure of the computation inside each behavior or a set of behaviors. This information is used to choose the right type and number of components for the system architecture.

9. **How do I discover the "computationally intensive" behaviors in my model ?**

   A straightforward approach is to produce bar charts for each leaf behavior in the model. For a reasonably complex design, the designer can use the hierarchical nature of the behaviors to display comparision between composite behaviors. Behaviors with low computation may be eliminated. For a behavior with high computation, the designer can display its child behaviors and so on. The author of the specification model can also supply this information upfront, since he or she would be well conversant with the model.

10. **Why should I evaluate an architecture before refinement ?**

    Most designs have constraints on execution time. The architecture exploration phase requires the designer to come up with the best set of components (and the distribution of computation over them) to meet this constraint. One way would be to generate the architecture model and then simulate it. This is time consuming if the designer has to go over several architectural choices. Evaluation of a model is a static analysis feature that allows the designer to check if an architectural choice meets the design constraints.

11. **If my architecture model simulation shows an encoding delay of "0.0ms", what did I do wrong ?**

This may be because the specification was not profiled before an architecture model was generated. Profiling generates information that allows architecture refinement to insert the appropiate delays for the target component.

12. **Can I refine any behavior in a model ?**

    The behavior which is set as the "top level" of the design is considered for architecture and communication refinement by the tools. Typically, the behavior representing the design under test (without the testbench) is set as the "top level" behavior. However, for RTL refinement, the designer may choose a particular behavior mapped to HW. This will allow the designer to examine only an interesting part of the design without having to simulate the entire model at cycle accurate level.

13. **Why do I need to rename all the generated models ?**

    Renaming is done to avoid overwriting of models during exploration. Automatically generated models are read-only for the same reason. Renaming also gives a suitable name to the model so that it can be easily recognized in the project window.

14. **I want multiple busses in my design. How do I map channels to busses ?**

    The design example in the tutorial has only one bus. The shortcut for mapping all channels to one bus is to map the top level behavior to that bus. In case of multiple busses, select Synthesis⟶Show Channels after allocating the busses. This would expose all the channels between the components. Individual channels can then be mapped to respective busses.

15. **Can I use point to point wire connections instead of busses in my design ?**

    Busses in SCE represent generic connection elements. It is possible to have point to point connections between components. This can be done simply by including such a point to point protocol in the protocol library and selecting it during communication synthesis. During channel mapping the designer must take care to map channels between only the relevant components to the point to point "connection element."

16. **Why do I need to do RTL preprocessing ?**

    Preprocessing is needed to generate a super finite state machine model of the design, which serves as an input to RTL refinement. The preprocessing step splits the behaviors into super states, with each super state comprising of a basic block.

17. **Why does RTL scheduling and binding display work only for leaf behaviors ?**

    During preprocessing each leaf behavior under the selected behavior for HW implemetation is converted to a super FSM. Displaying only one super FSM at a time

avoids overcrowding in the display and state name conflicts.

18. **How do I know which RTL units to choose ?**

    The designer chooses the RTL units that can perform the operations required in the model. RTL analysis gives statistical information on the number and type of operations in each super state. Structural constraints can put lower bound on the number of units. For example, if a unit with 3 inputs and 1 output is allocated, then atleast 4 busses must be allocated for feasible binding.

19. **How do I view source code generated by SCE ?**

    The SpecC source code for the behavior definition can be seen by clicking on the behavior in the hierarchy tree and selecting View⟶Source. The code for the behavior instance can be seen by right clicking on the instance in hierarchy and clicking Source. However, SCE also produces C, Verilog and Handel-C files. Since these files do not show up in the hierarchy, they have to be opened externally from a shell using standard editors.

20. **What is the current status of SCE ?**

    SCE is currently a demo version that works for select examples. In the future, it will be enhanced to a prototype tool.

21. **What other features are planned in the immediate future for SCE ?**

    In the immediate future, we plan to expand the libraries with more components, IPs and bus protocols. Improvements are planned for communication synthesis framework to handle complex communication architectures. There is also work planned for OS targetting and generation of RTOS models.

References

S. Abdi, J. Peng, R. Doemer, D. Shin, A. Gerstlauer, A. Gluhak, L. Cai, Q. Xie, H. Yu, P. Zhang, and D. Gajski, *System-on-Chip Environment - Tutorial*, CECS Technical Report 02-28, September 24, 2002.

A. Gerstlauer, R. Doemer, J. Peng, and D. Gajski, *System Design: A Practical Guide with SpecC*, Kluwer Academic Publishers Inc., June, 2001.

D. Gajski, J. Zhu, R. Doemer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers Inc., March, 2000.

D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*, Prentice Hall, June, 1994.

D. Gajski, F. Vahid, S. Narayan, and J. Gong, "SpecSyn: An Environment Supporting the Specify-Explore-Refine Paradigm for Hardware/Software System Design", IEEE Transactions on VLSI Systems, Vol. 6, No. 1, pp. 84-100, 1998, Awarded the IEEE VLSI Transactions Best Paper Award, June 2000.

D. Gajski, L. Ramachandran, F. Vahid, S. Narayan, and P. Fung, "100 hour design cycle : A test case", Proc. Europ. Design Automation Conf. EURO-DAC, 1994.