Laboratory Manual 1, MSPS

Introduction to Behavioral-Level Simulation

Description

The purpose of this laboratory exercise is mainly to provide the student with hands-on experience of the Cadence design tools used in the TSTE16 course: Mixed-Signal Processing Systems.

The laboratory also gives an illustration of top-down design methodology.

Outline

Document history

Rev	Date	Comment	Issued/Created by
1.0	2009-05-30	Creation.	Hägglund, et al.
2.0	2009-08-30	New document template, updated design kit, and changed the descriptions.	Per Löwenborg
PA3	2010-09-02	Aligned the document with the new Centos 5 system installed during Summer 2010. This also implies changing spectreHDL to veriloga.	J Jacob Wikner
PA4	2010-09-07	Removing some redundant blahblah due to change to veriloga. Changing the references to the matlab code.	J Jacob Wikner
PA6	2010-09-08	Updated with veriloga code	Niklas U Andersson
PA7	2010-09-08	Touch up and preparation for release	J Jacob Wikner
A	2010-09-08	Released for 2010 class.	J Jacob Wikner
P1B	2011-09-07	Touch up wrt. Cadence 6.	J Jacob Wikner
B	2011-09-07	Released for 2011 class.	J Jacob Wikner

1. OVERVIEW / INTRODUCTION

The principle system that is considered throughout the laboratory is shown in Figure 1.1. It consists of an analog filter, followed by a uniform sampling and then quantization. Only a very basic understanding of these blocks is required in order to complete this exercise and necessary details will be explained during the course of the laboratory.

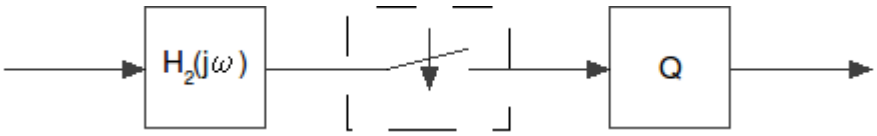

Figure 1.1: Basic system studied in the first laboratory exercises.

1.1 Initiating your Linux Environment

In this laboratory we will use a circuit design tool from the vendor Cadence. The Cadence tool suite contains many different tools, such as component simulators, integrated circuit layout tool, physical verification, logical verification and so on. For this lab, we will only consider the circuit-level simulation package (including a schematic capture program and a simulation environment).

When a circuit is created as a graphical netlist (schematic view), different types of circuit simulators can be used, for example HSPICE, Spectre, and SpectreS. The circuit simulators offer several types of analyses, such as DC, AC, transient, noise, etc. The speed and accuracy of the simulations also vary between the simulators. In this laboratory the Cadence Spectre simulator is used.

We start the exercise by setting up your linux environment. Load the course module:

```
module load TSTE16
```

Make sure that you are in your home account

```
cd
```

and then run a shell script

```
daisyCreateProj.sh TSTE16
```

> ✗ *Only run this script once –the very first time you work with this lab!*
> *Although low risk, you might risk loosing data if you run it on an existing project directory.*

This shell script will create a directory in your home directory called `TSTE16` (this is also what we from now on refer to as your `$WORKAREA` = `$HOME/TSTE16`). Further on, it will inside that directory create some other directories as well as links to files and directories. For now, you do not have to worry too much about them, but during the project work in this course, they will play a quite important role.

Source a dot file that was created by the script above:

```
source ~/.TSTE16_rc
```

(Please contact the lab responsible if this results in any errors or strange warning messages). Go to the `$WORKAREA` (`~/TSTE16`) by typing:

```
cde
```

Now, in the `$WORKAREA` directory you will find a link to `daisyProjSetup` and inside that directory you will also find lab simulation files, course information, etc. In your `$WORKAREA` you also find a `work_$USER` directory in which you should put all your personal files related to the lab.

✗ *Keep your files in a neat order! That is very important for project-based, top-down approach!*

2. GETTING STARTED WITH CADENCE

Launch the Cadence design framework by typing:

```
cad
```

For the lab and the project as such, you will not only need the Cadence tools but also a system-level simulation tool, either Matlab or Octave (GNU free ware). We are mainly going to use Matlab/Octave for modeling the digital parts of the transceiver design and for analyses of test data.

Once the program has started you will find a so called command-interpreter window (the `CIW`). This is the main interface window and here error messages, log files, warnings, etc., will be displayed. You also have access to the whole framework in the tools menu, etc. Dependent on your window manager settings, this window will raise if you press the `Alt + Home` key combination.

> ✗ *Notice that the Alt key is for some setups not activated. Contact the lab responsible or see the course blog for more information.*

Another very important window is the Library Manager which you launch/raise by pressing the combined keys `Alt + End` or through the menus `Tools > Library Manager`... in the `CIW`. The library manager should contain all the different design libraries that you need. We will also add our own libraries and circuit definitions through this tool.

Essentially, the only libraries required in these exercises are `analogLib`, and `ahdlLib`, and `daisy`. In `analogLib` we have ideal components like voltage sources, current sources, ideal resistors and capacitors, etc. The others contain more complex components. For example, in the `ahdlLib` library you find components that are described in `veriloga` (a language used for describing analog circuits).

> ✗ *You can copy and paste from the code of these components in order to build up your own cells.*

In the library `daisy` you find two cells named `daisyFileWriter` and `daisyFileReader` which are used later throughout the lab.

2.1 Creating a New Design Library

When you start a new project or a new set of design you need to create a new design library. In this library you can place your new cells. Select `File > New > Library` in the Library Manager. The tool asks you for where in the linux domain to put the library. Browse to your `$WORKAREA` and subdirectory:

```
work_$USER/oa/
```

where `$USER` is your login ID (e.g. `stdent123`). Further, enter a name for the new directory and press OK. In this lab we will refer to the following name:

```
tste16Lab1
```

✗ *Be consistent with naming, for example tag them with `tste16` or `group2` or something. so that it is clearly indicated what the purpose of the library is.*

After pressing OK, a new dialogue window appears where you should select a technology to attach to your library. Select `Attach to an existing techfile` and press OK. Choose Technology Library to `gpdk045` and press OK again.

Now you have a brand new design library in which you can start creating your own designs. As a companion to the design library we will now create yet another library in the same way. Place it in the same linux area and call it:

`tste16Lab1Test`

or `<SAME_NAME_AS_YOU_USED_BEFORE>Test` so that the postfix is `Test`. This library is going to contain your test benches.

✗ *It is not good practice to have test benches in your actual design library. They should be put in a separate design directory and should preferrably also be technology independent.*

2.2 Creating a New Cell View

In a design library several cells can be stored (compare with the library structure in a computer), but each cell can also consist of several different descriptions (cell views). For example, a circuit can be described by a short code snippet, a circuit netlist, a layout description, and so on. (In fact one can even add documentation, web pages, pdfs, etc. as cell views too.) When you are creating a new cell you therefore also need to choose which type of description you will use. A cell/cell view is created by the following procedure:

Select the directory where you like to have your cell view. Select `File > New > Cell View`. Enter a `Cell Name` and choose a `Tool`. Be consistent when it comes to naming of cells too (this is very vital for e.g. the design projects and sharing cells between different team members). One wants, for example, to avoid names like: `cell1` or `filter`. Instead use a descriptive name and also tagged with your project name, like for example:

`tste16Lab1RcFilter`

or something like that.

✗ *It is good design practice to tag the cell name with the library name too. Yes, cell names might become long, but in larger projects it is worth the pain.*

The tools that are available for this lab are:

`Composer-Schematic` to design a circuit in the schematics mode, i.e., a netlist of for example resistors, capacitors, transistors, and so on.

`Composer-Symbol` to create symbols for a schematic to be used in hierarchical designs.

`VerilogA-Editor` to write a high-level description of a specific cell. This can have the advantage that a high-level model can be used for determining specification of specific sub-blocks as well as decreasing the simulation time of a larger project.

`Hierarchy Editor` to select which view that is used for a specific simulation. This simplifies the top-down design approach, since the implementation of a specific system can be decomposed into subblocks which can gradually be refined.

`Virtuoso` to create layout of a circuit.

The tool you have selected will then be loaded and it is now possible to start designing your circuit.

2.3 Schematic View

The schematic composer is a graphical interface to design circuits schematics. An example of a schematic view is shown in Figure 2.1.

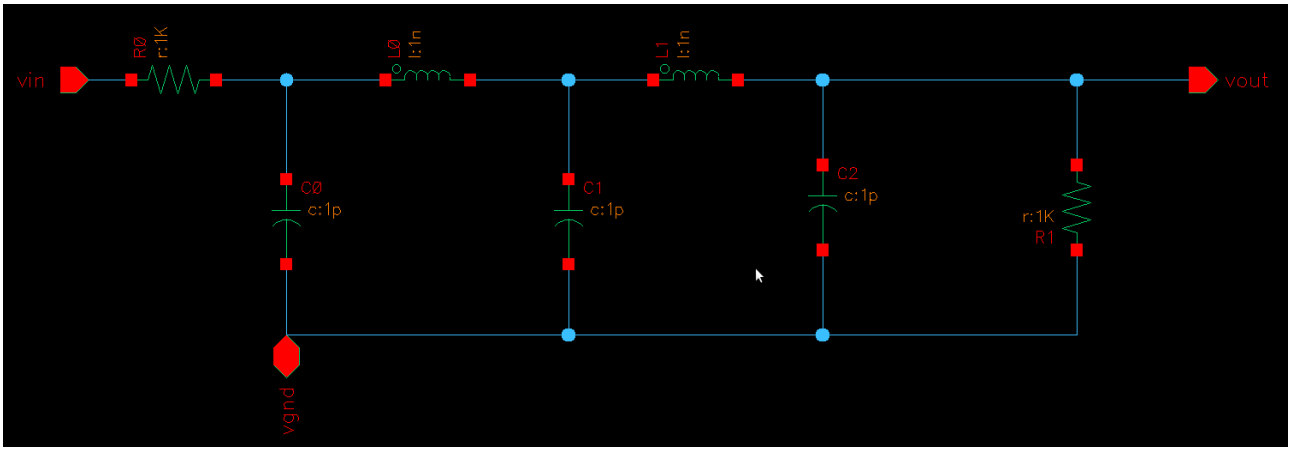

Figure 2.1: Schematic of an RLC type of filter containing input and output ports.

The tool can internally generate a netlist in text format describing the circuit topology. This netlist is normally used for simulation to evaluate the performance or test the functionality of the circuit. In this particular case we have only one schematic view, which is more or less an ideal description of the circuit. No parasitic wire capacitors or resistors are found in this model, since those parasitics would be dependent on the layout of the circuit instead. Post-layout simulations can indeed also be performed so that further include effects of parasitic elements, and thereby come even closer towards a description of the real-world implementation. This is however out of the scope of this lab.

2.3.1 Add a New Instance to the Schematic

Elements such as voltage sources, transistors, and capacitors are normally predefined building blocks that can be inserted in the schematic either through `Add > Instance` or by the shortcut key '`i`'. A popup window will appear where you can add a component by specifying the library and cellview. You can either add a predefined cell, for example those listed in Table 2.1, or components that you have developed yourself. Normally, only cell views of type symbol are inserted into a schematic.

Component	Library	Cell	Short-cut Key
Supply	analogLib	vdd	ctrl + shift + d
Ground	analogLib	gnd	ctrl + shift + g
DC voltage source	analogLib	vdc	ctrl + shift + v
DC current source	analogLib	idc	ctrl + shift + i
Sinusoidal voltage source	analogLib	vsin	N/A
Sinusoidal current source	analogLib	isin	N/A
Pulse voltage source	analogLib	vpulse	N/A
Pulse current source	analogLib	ipulse	N/A
Voltage controlled voltage source	analogLib	vcvs	N/A
Ideal resistor	analogLib	res	ctrl + shift + r
Ideal capacitor	analogLib	cap	ctrl + shift + c
Ideal inductor	analogLib	ind	ctrl + shift + l
NMOS transistor	analogLib	nmos4	ctrl + shift + n
PMOS transistor	analogLib	pmos4	ctrl + shift + p
Dump information to file	daisy	daisyFileWriter	
Get information from file	daisy	daisyFileReader	

Table 2.1: List of cells that are useful for this lab.

When a component is to be placed in the schematic, either by writing the names of the library and the cell view or by using the Browse option, the properties of the selected components will appear further down in the pop-up window. For example, the width and length of a transistor can be defined, the voltage of a voltage source, resistance, etc.

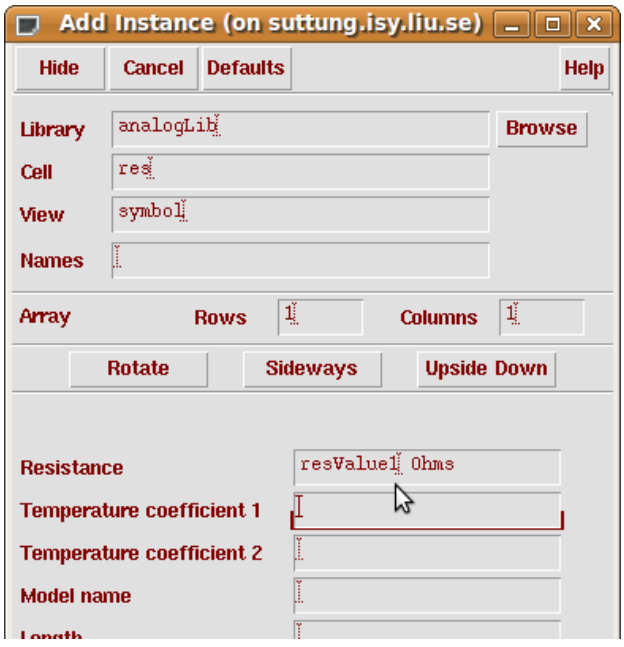

Figure 2.2: Illustration of how to add a design variable name.

2.3.2 Defining Design Variables

During the circuit design procedure it is common that, for example, the capacitance of a capacitor, bias currents, and voltage values must be adjusted to ensure that all transistors are operating in their desired operation region and to meet a certain specification. In this case, it is easier to define variables for parameters that you frequently need to adjust since you will get access to them directly from the simulation tool. Defining variables is done by simply writing a variable name in the desired field of the specific a component. Either add the variable name once you instantiate your component or do it

afterwards by clicking on the component and press 'q'. This will give you a properties window, identical to the window used when instantiating the component. In for example a resistor, the resistance can be set to the e.g. the variable `resValue1`.

See Figure 2.2 for a reference (notice that Cadence 6.x has a slightly different look-and-feel).

> ✗ *In a shared project, it is good practice to tag the variable with your signature. For example:* `resValue1` → `jjwResValue1`.
> *The reason for this is that once other designers invoke your design into their simulator test benches the risk for using identical variable names is minimized and it also becomes easier to identify and then to ask designer 'jjw' for a proper variable setup. Please remember this for your future project assignments.*

2.4 Symbolic View

In the symbolic view a symbol for the cell is created. This symbol is used in order to increase the level of abstraction by using a hierarchical definition of the circuit. For example, the netlist of the ladder filter shown in Figure 2.1 can be represented by the symbol in Figure 2.3 so that it can be used on a "higher" level of hierarchy. Typically, several instances of the same building block are used. Hence, creating a symbol for a specific circuit is most often very practical.

The block can be made more pretty by adding some additional text labels. Remove the `[@partName]` and `[@instanceName]` labels by clicking on them and press delete. Then press `alt + r` and `alt + a` to add some new labels.

> ✗ *Once again, to fix the missing Alt key, talk to lab assistant.*

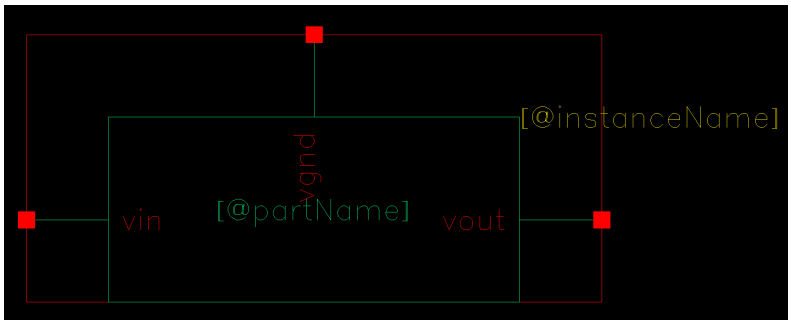

Figure 2.3: Symbol view of the schematic cell.

2.5 Functional View (veriloga)

Describing a building block by the use of a hardware description language (HDL) can be very handy. Especially when designing large systems where the complexity is so large that it becomes impossible to verify the functionality on a transistor or RC netlist level. This is due to that a functional description of a building block most often can rapidly be written. This has the benefits that the system can be tested

at an early stage in the design process which leads to less redesign cycles. Further, the simulation time can also be decreased by simulating the system using several different levels of abstraction. There are drawbacks, such that the model is somewhat "incorrect" and that it therefore does not capture vital flaws or misbehavior of the corresponding netlist implementation of the same block.

The veriloga language is a programming language which can be interpreted by the Cadence Spectre simulator. This has the benefits that several different descriptions of the individual building blocks can be simulated at the same time. For example, the ladder filter shown in Figure 2.1 can be simulated with a veriloga description of its behavior. The syntax in the veriloga language is closely related to the verilog RTL programming language. More about this language is given later in the lab manual.

2.6 Hierarchy Editor

The hierarchy editor is used for dealing with several different descriptions of the building blocks within a design. For example, the filter building block can be described by the transfer function as a mathematical expression in a first model. In the next model, the filter can be described by ideal components such as resistors, capacitors, and buffers. Later on, these components can be modelled as non-ideal components which more corresponds to the actual implementation of a fabricated chip. Hence, there can exist a large number of descriptions for the same building block. The hierarchy editor can be used to switch between the different descriptions of the building blocks, which facilitates the top-down design procedure.

Another great advantage is that the hierarchy editor enables us to simulate the same cell with different views if there are several entities instantiated in the test bench.

2.7 Simulating a Circuit Schematic

As stated earlier, the simulation environment can use several different types of simulators. The functionality, i.e., the types of analyses, the speed, and the approximation methods when solving the systems of differential equations, differ between the simulators. Analyses that can be used are, for example, large-signal analysis (DC), small-signal analysis (AC), transient analysis, different type of noise analyses, distortion analysis and sensitivity analysis. In the labs we will use the three most commonly used analyses namely DC, AC and transient analysis

The simulation setup can be divided into three parts namely the analysis setup, setup of the output signals (the ones we want to plot, observe and extract information from), and adding design variables to the simulation.

2.7.1 Analysis Setup

DC:

The analysis part is like the ones in Spice: you can either check the DC behavior of the circuit and check that your components are operating correctly, i.e., all transistors are operating in the correct operation region and so on. Here you can for example sweep a parameter like the bias current in your amplifier, supply voltage, etc., to further check the operating conditions.

AC:

The properties of the circuit with respect to a small change in the DC operating point can be evaluated by linearizing nonlinear components around their DC operation points. The linearzation as well as the simulation is performed by the AC analysis.

Transient:

The transient analysis is used when the time response of a circuit is of interest. This analysis method takes into account clipping of the circuit and other large signal effects.

Parametric analysis:

Parametric analysis are used when two or more parameters are to be swept independently of each other. In this case sweep the current in the DC analysis and add a parametric sweep using the menu alternative in the Affirma Window

```
Tools > Parametric Analysis
```

Enter the name of the variable for the output transistor width into the field Variable Name. Add the ranges of the sweep and start the analysis by selecting

```
Analysis > Start
```

2.7.2 Defining Output Signals

The outputs of the simulation can, for example, be the output voltage from an amplifier, the current through a current mirror or a mathematically defined function to calculate the unity-gain frequency of a circuit. These outputs can be selected from the schematic by using the menu command `Output > To Be Plotted > Select In Schematic` and then select the output to be plotted each time you do a simulation. Note that by selecting a wire, the node voltage for that node will be plotted, by selecting a node (for example on one of the resistor terminals) the current through that node will be plotted.

2.7.3 Defining Design Variables

The variables defined in the schematic view can be imported into the simulation environment. This is done by using the menu alternative

```
Variables > Copy from Cellview
```

All the variables will now appear in the variable field in the Affirma Circuit Design Environment window. To assign a value to one of the parameters just double click on the variable and enter the desired value.

Notice that you an write a mathematical formula here as well as you can let a design variable be a function of another design variable.

3. INSPECTING SIMULATION RESULTS

The DC operation conditions of a circuit are computed when a DC simulation is performed. If you like to display the result directly in the schematic use the menu alternative

`Result > Annotate >`

If you like to display a complete list of the operation condition use

`Result > Print >`

Some different alternatives in the pop-up menus are listed in table below with a short explanation.

Menu alternative	Description
DC node voltage	The voltage in a specific node or all nodes.
DC operating point	Operating point of a device with for example power consumption.

Table 3.1: Print results options

3.1 The Calculator

The calculator can be used to, for example, add, subtract, multiply, or take the ratio between two waves, calculate the discrete Fourier transform (DFT) or efficiently compute the DC gain, unity-gain frequency, phase margin and slew rate of a circuit. It is very handy when you are trying to increase the performance of a circuit.

Plotting the derivative of a wave can be done in the following way: Start the calculator from the tool bar in the Waveform Window. Press the `Wave` button and select a wave in the Waveform Window. Click on the `Special Functions` selection box and choose the `deriv` command. To plot the derivative just select the `Plot` button.

Some practical commands to use are compiled in Table 3.2 for convenience.

Command	Description
`phase`	Computes the phase of the output
`phaseMargin`	Computes the phase margin of the circuit
`cross`	Returns the x value at which the waveform crosses a certain y value
`mag`	Displays the magnitude response of the wave (linear scale)
`dB`	The quantity expressed in decibel
`value`	Returns the y-value for a certain x value

Table 3.2: Example on some useful calculator commands

✗ *In addition one can add scripts, etc., and commands through the CIW. All data available in the calculator is also available in the CIW. You can for example copy the expression from the calculator window and paste it into the CIW and press return, etc.*

✗ *Experienced users do not use the calculator that much, but instead writes so called SKILL scripts to calculate important parameters. The calculator is however a very good starting point for newbies.*

4. USEFUL SHORTCUT KEYS IN THE SCHEMATIC VIEW

There are many shortcuts (bind keys) in the program to speed up the design process (there are some built-in ones, but local bind keys can also quite easily be defined). Some of the most useful bind keys are listed in table 4.1.

There are some key and mouse combinations that can be used too. For example, during a move and copy command, a double click at the middle mouse button will make it possible to rotate and flip as well as coping an object several times.

Short-cut	Description
`B`	Return from hierarchy descent
`c`	Copy. Press c then choose object to move
`delete key`	Delete
`E`	Descend in the hierarchy
`escape`	End last command
`f`	Zoom out to full view
`i`	Insert component
`l (small L)`	Add a label to a wire
`m`	Move an object
`p`	Add a pin (needed only for hierarchical objects)
`q`	Properties of an object
`r`	Rotate an object
`u`	Undo
`w`	Add a wire
`X`	Save the cellview
`z`	Zoom in a box
`F1`	Help
`F6`	Redraw
`8`	Zoom out
`9`	Zoom in

Table 4.1: Shortcut keys (bind keys)

5. THE FIRST MODEL OF FILTER AND ADC CHAIN

The task of this first lab is to design a filter (anti-aliasing filter) and ADC chain. In this section a simple model of this chain is designed. The filter should be a first-order filter with the -3-dB cut-off frequency at 100 kHz and the ADC should have 16 quantization levels, i.e., it should be a 4-bit converter.

We start with the design of the filter which we would like to implement using the veriloga language. Typically, in order to make the veriloga cell view it is advantageous to start to make the symbol for the filter. The filter symbol should consist of three terminals; input, output, and a ground terminal.

In section 2.1 and 2.2 it was shown how to add a new library and cellview. So here, the first thing to do is either to create a new library or chose to use the one you created before. A good name for the library is `tste16Lab1`. Next step is to add a symbolic cellview to that library by using the library manager. Use for example the following name: `tste16Lab1RcFilter`.

> ✗ *Once again, it is good practice to also inherit the library name in the cell name. This is for compatibility with other tool vendors, where the design libraries are treated a bit differently, and also in a spice netlist file, the hierarchy does not exist.*

5.1 Drawing the Symbol

When you have created the cell, an empty window called `Virtuoso Symbol Editing` appears. In this window we will create the symbol shown in Figure .

Start by inserting the inout and output pins by pressing the shortcut for insert pin, `p`, and add the pins `Vin`, `Vout`, and `AGND`. (You can separate the pin names with space on a single line). Note, that the pins should be of direction `inputOutput` type `square`.

Add the shapes of the symbol, i.e., rectangle, line, and arc from the menu `Add > Shape`. The last part is to add a selection box around the symbol. This is done by the command `Add > Selection box` and then press Automatic. Your symbol could look like the one in Figure 5.1.

> ✗ *Use "good"signal names and be consistent. Normally in projects there should be strict requirements on signal names in order to avoid confusion when other designers are working with "unknown"blocks. There must not be any ambiguity when it comes to understanding the signal and its purpose.*

One can pretty print it even more, by adding labels, putting pins on grid, etc. If you like you can do this by first deselecting everyting (press in the empty black area outside the symbol). Then press `alt + a` to get simulation annotation variables on the pins, `alt + r` to get cell names, etc. Finally press `ctrl + a` to select everything and then `ctrl + 2` to put it on grid. Move labels around if you like.

Check and Save the symbol, i.e., `Design > Check and Save`. Make sure that there are no errors or warnings in the CIW. If there are any errors correct them before you continue.

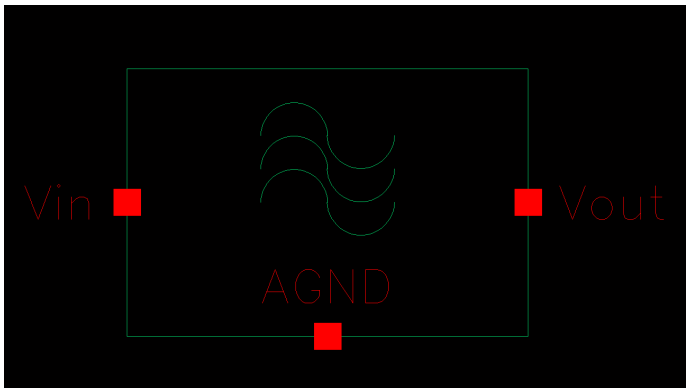

Figure 5.1: Symbol view of the filter.

The symbol is now finished and the next step is to create the functional description of the filter using veriloga. We should now create a veriloga cellview. This can be done directly from the Affirma Symbol Editing window by using the command

```
Design > Create Cellview > From Cellview...
```

In the dialog box that appears change the `Tool / Data Type` to `Verilog-Editor`. Change the view name from `functional` to `veriloga`. A new dialog box might appear to ask you if you are sure about the change, choose OK. An emacs buffer will appear with some automatically generated lines of code, which should look like the code in Table 5.1.

In the automatically generated code a new building block (entity) is defined by the `module` command and the input and outputs are defined. The direction of the ports are defined and the ports are given the `electrical` property. The module definition is finally ended by the `endmodule` command.

```
// VerilogA for tste16Lab1, tste16Lab1rcFilter, veriloga

  `include "constants.vams"
  `include "disciplines.vams"

  module tste16Lab1rcFilter(AGND, Vin, Vout);

        inout AGND;
        electrical AGND;
        inout Vin;
        electrical Vin;
        inout Vout;
        electrical Vout;

  endmodule
```

Table 5.1: Code template when creating veriloga view from scratch.

Now, in the first version, the filter should be modelled as a single pole filter. For a rapid implementation we should implement this pole using the command `laplace_zp`. Further, we like to be able to move the pole without changing the veriloga code, i.e., we like to use an instance parameter for the pole position.

A code example for the `tste16Lab1RcFilter` with a single pole is given by (pay special attention to the code structure):

```
// VerilogA for tste16Lab1, tste16Lab1rcFilter, veriloga

`include "constants.vams"
`include "disciplines.vams"

module tste16Lab1rcFilter(AGND, Vin, Vout);
inout AGND;
electrical AGND;
inout Vin;
electrical Vin;
inout Vout;
electrical Vout;

parameter real pole1 = -100e3*2*3.1415 from (-inf:0);

analog begin

       V(Vout,AGND) <+ laplace_zp(V(Vin,AGND), {},{pole1, 0});

end

endmodule
```

Table 5.2: Code illustrating the use of the `laplace_zp` command.

The variable type (`real`, `integer`, `string`,...) for each parameter must be declared. In this case, we have only one parameter which indicates the pole of the filter. *Notice the sign!* Further, it is also advantageous to give a default value of the parameters as well as a region which the parameter must be within. We also have a possibility to define local parameters that can be used in the model.

The `analog` module is used for the large-signal behavior of the circuit. This is the core description of the module. The `analog` module is typically a mathematical mapping from its input to its output. For example:

```
value_quantity( node1[, node2] ) <+ expression;
```

Here, we also have the possibility to use, e.g., `if` statements etc. The `analog` block starts with `analog begin` and is ended by the `end` command.

In the example above we have an expression which performs Laplace transformations. It actually states that. `V(Vout)`, i.e., the voltage at the output node, is set to the inverse Laplace transform of the product between the Laplace transform of the input voltage and the frequency transfer function. The `zp` stands for that the two components to the right of the expression are a list of zeros and poles, respectively. The part which is described by `{pole1, 0}` says that the pole is real since the imaginary part is zero. The empty curly brackets `{}` says that no zeros exist.

For more information about the Laplace command please look at the reference manual. This can be found from the `Help > Cadence documentation` in for example the schematic editor and then select

veriloga reference. There is also a lot of information available on the internet and quite often much more accessible than from Cadence.

5.2 Setting up a Test Bench to Simulate the Filter

Any circuit must be simulated in order to validate its functionality. In this case we must check that it has a proper transfer function. In order to do this simulation, a test bench must be designed. It should be able to evaluate the transfer function of the filter.

Previously, we created a library with the `Test` extension, in our example it was `tste16Lab1Test`. In that library create a schematic cellview which is called

`tste16Lab1RcFilter_T`

> ✗ *It is good design practice to also let the test bench reflect the name of the cell that is tested!*

The test bench should look like the one in Figure 5.3, i.e, the input to the filter is an input voltage source while the output is loaded by a capacitor.

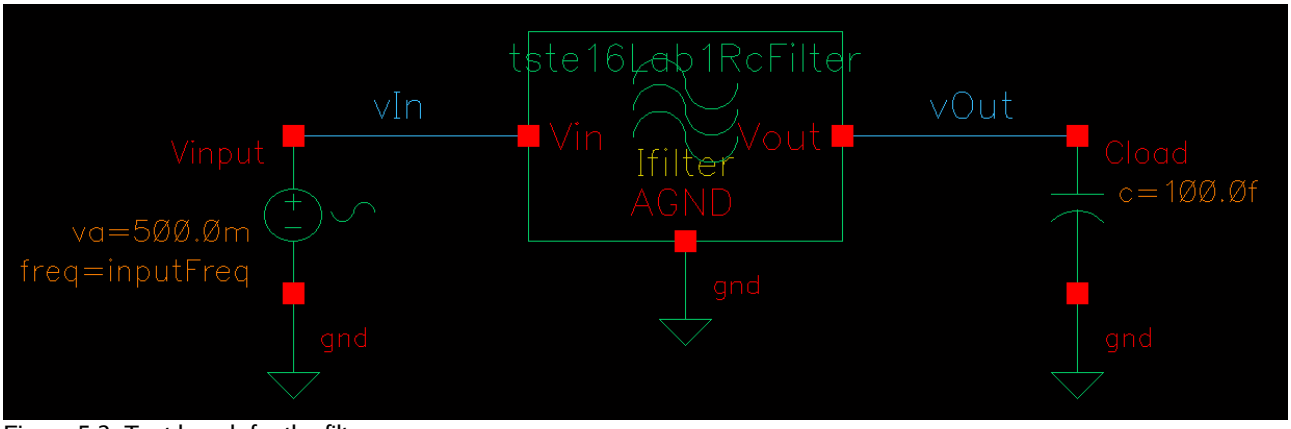

Figure 5.2: Test bench for the filter.

Start to add the instance for the filter building block by `Add > Instance` or by the shortkey '`i`'. Fetch the `tste16Lab1RcFilter` component from your `tste16Lab1` design library. Use the `symbol` view name - not the `veriloga`. Use the `Browse` button or type in the missing fields. Place this building block in the test bench. Continue with the components `analogLib/vsin` for a sinusoidal voltage source, `analogLib/gnd` for a ground potential, and the `analogLib/cap` for the load capacitor.

> ✗ *Here it is good place to mention that magic Escape button. Cadence nests commands and sometimes you will notice it does not do what you like... Probably you have an old command call active (see bottom of the window –in the status bar). This command is canceled with the Esc button. You will see that even experienced Cadence users hits the Escape button as a reflex now and then.*

The parameters for the components can be set directly before the components are placed in the schematic or by selecting the building block afterwards and pressing `q`.

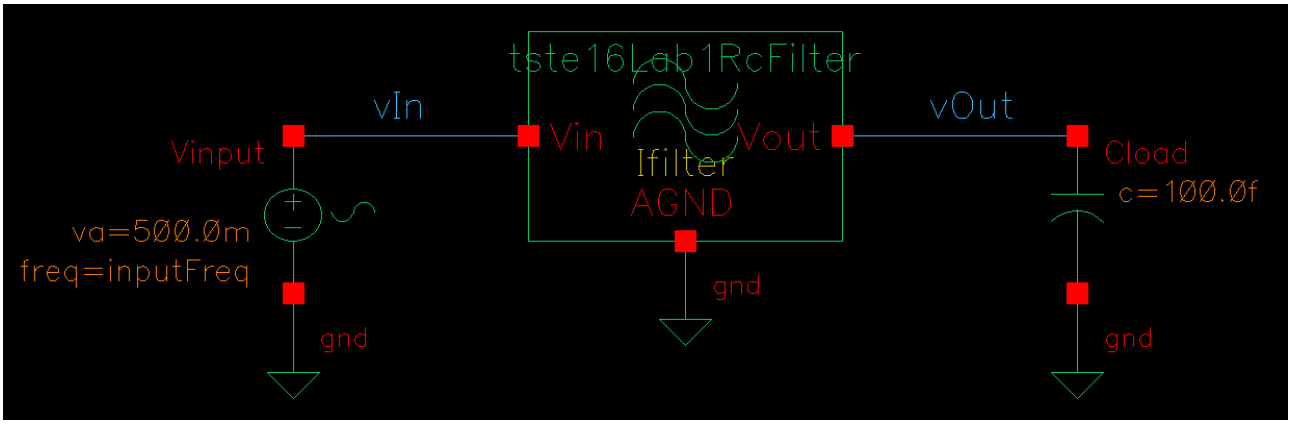

Figure 5.3: Test bench for the filter.

> ✗ *It is good design practice to name every component in your design. For example, use the name `Cload` for the capacitance, `Vinput` for the source, and `Ifilter` for the filter. The name is changed by pressing `q` on the component and change `Instance Name`.*

Change the load capacitance from the default value of 1 pF to 0.1 pF. For the filter, i.e., the cell `tste16Lab1RcFilter`, press '`q`' and in the selection box `CDF Parameter of View` choose `veriloga`. Now, the parameter `pole1` appears in which you can specify a value or a parameter. In this case set the parameter `pole1` to the simulation variable `firstPole` (the pole location is given in rad/s). You can also choose to display the settings in the schematic by using the `Display` selection box.

Enter the parameters found in Table 5.3 in the sinusoidal voltage source properties window. Add labels to the input and output wires by using the bind key '`l`' (lower-case L). Tip in the name and place the label on the wire.

Finally save the test bench and all subcells by pressing `shift + y`.

Parameter	Value	Explanation
AC magnitude	1	Using 1 V gives the transfer function at the output.
AC phase	0	The phase of the AC sinusoid in the frequency domain
DC voltage	1.65	The DC input voltage to the filter.
Amplitude	0.5	The amplitude of the sinusoidal waveform in time domain
Frequency	`inputFreq`	The frequency of the sinusoidal waveform in time domain [Hz]

Table 5.3: Input source settings for the filter test bench.

5.3 Simulating the Filter

To start the simulation environment use the menu alternative `Launch > ADE L` from the `tste16Lab1RcFilter_T` schematic editing window. A new window called Virtuoso Analog Design Environment (ADE) appears.

In this window we first get the design variables from the schematic by `Variables > Copy From Cellview`. The variables show up in lower left box. Double click on them and set them to the values found in Table 5.4.

Variable	Value	Unit
`firstpole`	100k * 2 * 3.14159	rad/s
`inputFreq`	50k	Hz

Table 5.4: Simulator variable settings.

Continue to add the output that you like to have from the simulation. This is done by the command `Outputs > To Be Plotted > Select On Schematic`. Select the output and input nodes and end the sequence of selection by pressing Escape. If you are selecting a wire the voltage of that wire will be displayed, the current can be displayed by choosing a node.

Before we run the simulation, we need to choose the type of analysis. This is done by:

`Analyses > Choose ...`

First we add a DC analysis. Select the `dc` analysis button in the appearing window and tick the `Save DC Operating Point`. In this case we do not like to have any type of sweep so click on the `Apply` button.

To compute the small-signal transfer function an `AC` analysis must be performed. Select the `ac` analysis button and since we like to do a frequency sweep we just enter the start frequency for example `1` and the stop frequency at `1M` and then click Apply (i.e., the input frequency is swept from 1 Hz to 1 MHz).

Before the simulation can be started we have to make sure that the cellview type `veriloga` is supported for simulation. This is done by checking that the `veriloga` view exists in the fields `Switch View List` and `Stop View List` in the `Setup > Environment` menu. Type `veriloga` at the end of the list in these fields if they are not already inserted and press OK. Note, that this procedure must be performed for every new schematic you are about to simulate.

The definition of the simulation is now complete. The simulation is started by clicking on the traffic light with green light. Possible errors will be shown in the `CIW` and in a `spectre.out` window that also appears. One very common cause of error is that you have not saved the schematic properly, one can then go back and do a `ctrl + y` again.

Check the -3 dB bandwidth of the filter. This is done by double clicking on one of the waveforms from the AC simulation. In the dialog box select both signals, `/vIn` and `/vOut` and select `dB20` as the scale followed by an OK. The waves are now shown in a logarithmic scale instead. Measure the -3 dB frequency by tracking the curve with your mouse (you see the x and y values at the top of the window). You can use markers (bind keys `A` and `B`) to "mark" the place of -3 dB. Instead of double clicking, you can access the dialog box from the menu `Curves → Edit` in the waveform window.

What is your measured -3 dB frequency?	

Change the frequency of the first pole to 25 kHz and resimulate the circuit with a transient simulation.

What is the amplitude of the output signal?	
How can you determine it from the AC simulation results?	

5.4 Analog-to-Digital Converter (ADC)

After the filter, an analog-to-digital converter (ADC) converts the analog and filtered signal into a digital representation. We here want to design a high-level functional model of the ADC. The ADC operation consists of two main operations: uniform sampling and quantization. In practice the analog signal does not only need to be sampled, but also held constant after sampling in order to facilitate a quantization that works in practice. We therefore design and model one sample-and-hold (S&H) circuit and one quantization block (Q) to form the ADC.

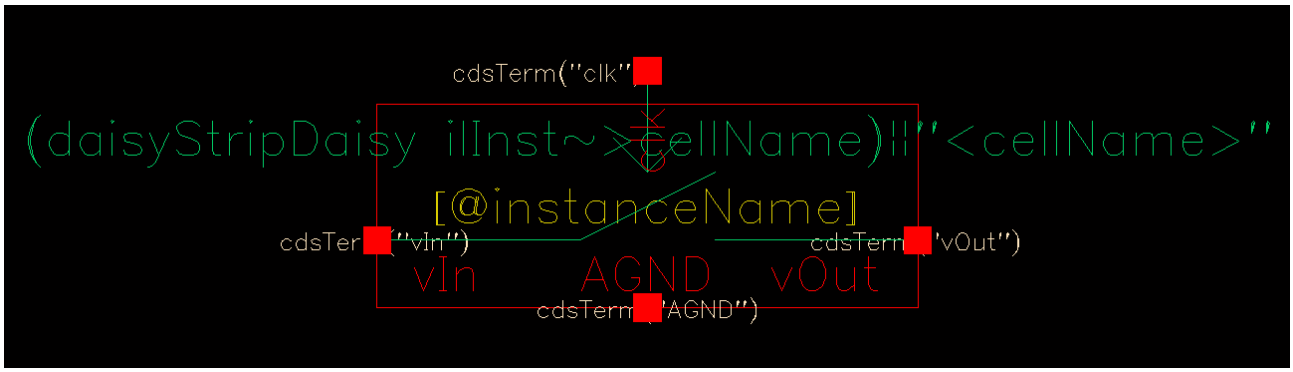

Figure 5.4: Example of S&H symbol.

We start by making a symbol for the sample-and-hold circuit. It needs one input terminal, one output terminal, one clock terminal, and one ground terminal. Go to the library manager (`alt + end`) and create a new symbol cell view named `tste16Lab1SampleHold` in your `tste16Lab1` library. An example of the resulting sample-and-hold symbol is shown in Figure 5.4. Terminal labels, etc., have been added according to the instructions given when designing the filter symbol.

5.4.1 Sample-and-Hold

A first behavioral-level model of the sample-and-hold circuit can now be written using veriloga. Create a veriloga view from the symbol editor. Do not forget to select `Verilog-Editor` in the `Tool / Data Type` submenu and change `functional` to `veriloga`. You will now have the following generated code:

```
// VerilogA for tste16Lab1, tste16Lab1SampleHold, veriloga

`include "constants.vams"
`include "disciplines.vams"

module tste16Lab1SampleHold(AGND, Vin, Vout, clk);

        inout AGND;
        electrical AGND;
        inout Vin;
        electrical Vin;
        inout Vout;
        electrical Vout;
        inout clk;
electrical clk;

endmodule
```

The sample-and-hold circuit will be triggered by a clock signal (`clk`). It is common that the sample-and-hold output signal is either following the input signal (track-mode) or holding a previously sampled value (hold-mode). This depends on the length of the hold phase. Certain architectures, like for example switched-capacitor based S&H circuits have no tracking phase and therefore hold the sampled value until a new one is acquired. This latter approach is what we choose to model here.

Let a rising clock edge define the acquisition of a new sample value. In veriloga, this can be done by comparing the clock signal with a threshold level. Add a real parameter called `clkThreshold` with nominal value 1.65, and with values restricted to the open interval `(0, inf)`. This is the threshold level equal to half the power supply voltage, 3.3 V. Then define a local variable, `vOutTemp`, that stores the sampled signal values. The sampling instant can now be defined by

```
@(cross(V(clk,AGND) - clkThreshold,1))
              begin
                    vOutTemp = V(Vin,AGND);
              end
V(Vout,AGND) <+ vOutTemp;
```

The statement `@(cross(V(clk,AGND) - clkThreshold,1))` detects when the voltage between the two terminals `clk` and `AGND`, i.e., `V(clk,AGND)`, becomes larger than `clkThreshold` on a rising edge (hence the `1`). In that case, `vOutTemp` stores the input voltage. Finally, let the output voltage be assigned to the value of `vOutTemp`.

Tip! Look through the `ahdlLib` in your library manager for more examples on parameter definitions as well as assignment syntaxes.

A basic veriloga description of a sample-and-hold circuit is now ready to be incorporated with the RC filter. Create a new test bench, i.e., a schematic cell view in the `tste16Lab1Test` library, called `tste16Lab1RcFilterSH_T`. Add an instance of your filter and one of the sample-and-hold. Name them during this operation too.

Further, add a sinusoidal input voltage source and a capacitor at the output of the sample-and-hold. Also add a pulse source, `vpulse`, from `analogLib` which needs to be grounded and connect it to the CLK terminal of the `tste16Lab1SampleHold` symbol. In the object property list of the pulse source, let `Voltage 1` equal 3.3 V, `Voltage 2` equal 0 V, and add two new parameters, `clkPeriod` and `tRiseFall`, which are the clock period and the rise and fall time of the clock pulse. Let `Rise Time` and `Fall Time` be equal to `tRiseFall`, `Period` equal to `clkPeriod`, and `Pulse Width` equal to `clkPeriod/2`. Connect the sample-and-hold symbol between the filter and the load capacitor.

It is now time to simulate the filter and sample-and-hold circuit. Open the ADE, perform a transient analysis and study the sampled-and-held output signal. Let the clock period be one tenth of the input signal period and the rise/fall time be one tenth of the clock period. Choose a reasonably long simulation time. Compare the output waveform with the input and make sure that you understand the operation of the S&H circuit.

5.4.2 Four-bit Quantizer

In order to obtain a digital representation of the sampled data, the amplitude needs to be quantized. Hence, the sampled-and-held data is represented with an N-bit binary number. We will here consider a so called flash ADC, which is conceptually the simplest ADC architecture. In an N-bit flash ADC, the quantization of the input is carried out by comparing the input with $2^N - 1$ (ideally equally spaced) different reference levels. A symbol for a 4-bit quantizer is shown in Figure 5.5.

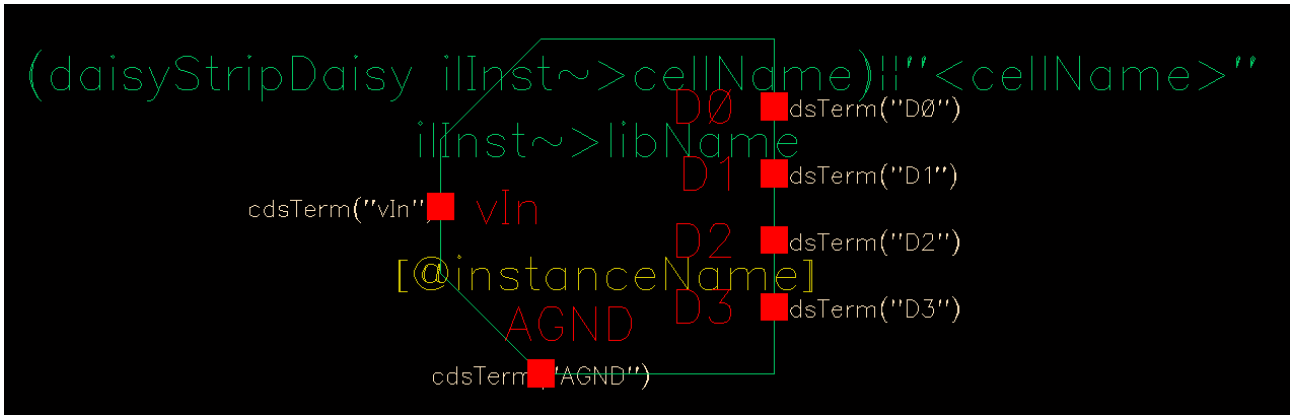

Figure 5.5: Example of a quantizer symbol.

We here consider a relatively small (few quantization levels) flash ADC which is typically used in a so called pipelined ADC architecture. The pipelined ADC is not discussed further in this laboratory.

Start with creating a symbol cell view named `tste16Lab1Quantizer` like the one shown in Figure 5.5. Then create a veriloga view. A 4-bit converter requires 15 different reference levels. In the behavioral-level description, it can be useful to declare local variables for the reference levels and the output. The variable holding the reference levels, `qLevel[],` can be defined as a 16 bit wide vector where each bit of the vector can be assigned as e.g.,

```
//Reference levels

        qLevel[15] = vMin+15*vRef/16.0;
        qLevel[14] = vMin+14*vRef/16.0;
        qLevel[13] = vMin+13*vRef/16.0;


        qLevel[1] = vMin+1*vRef/16.0;
        qLevel[0] = vMin+0*vRef/16.0;

//Output
real output[3]=0; output[2]=0; output[1]=0; output[0]=0;
```

`vMin` is the minimum input signal level and `vRef` is the input signal range. Note that the lowest index of the arrays defined above is 0. Hence, the first element of `qLevel` is accessed with `qLevel[0]` and the last element with `qLevel[15]`. The `output[]` vector is the quantized digital output (binary offset). The vector assignment can of cource be done with a `for` loop as well, e.g.,

```
//Reference levels

integer i;
        for (i = 0; i < 16; i = i +1)
                begin
                        qLevel[i] = vMin+i*vRef/16.0;
                end
```

In the quantization process, the input should be compared with the reference levels. This can be done with a set of `if` statements, e.g.,

```
if ((V(vIn, AGND) >= qLevel[10]) && (V(vIn,AGND) < qLevel[11])
begin
        output[3] = 1;
        output[2] = 0;
        output[1] = 1;
        output[0] = 0;
end
```

Finally, we need to assign the output voltages their proper values, e.g.,

```
V(D0) <+ vLow*(1 - output[0]) + vHigh*(output[0]);
```

where `vLow` and `vHigh` are the voltage levels (parameters) of the output data 0 and 1, respectively. The assignment above results in non-continuous output voltage wave-form. If a more practical output signal format is desired, non-zero transition times can be incorporated as

```
V(D0) <+ transition(vLow*(1-output[0])+vHigh*(output[0]),td,tr,tf);
```

where `td` is the delay time, `tr` is the rise time, and `tf` is the fall time of the output data.

> ✗ *Tip! For debugging purposes the $strobe command can be used. The $strobe command can for example return parameter values during simulation.*

> ✗ *Tip! In order to reduce computational load during simulation ,the @(initial_step) command can be used. This command tells the simulator only to compute the following section, enclosed by begin and end, **once** at startup, and not every time the module is accessed.*

The veriloga describing the 4-bit quantizer could now look something like the snippet below. Notice that this is just an example and several ways can be used.

```
// VerilogA for tste16Lab1, tste16Lab1Quantizer, veriloga

`include "constants.vams"
`include "disciplines.vams"

module tste16Lab1Quantizer(AGND, D0, D1, D2, D3, vIn);

inout AGND;
electrical AGND;

etc.. (the rest of the port declarations)

parameter real vLow  = 0 from [0:inf);
parameter real vHigh = 3.3 from [0:inf);

etc … (the rest of the parameters)

// Reference levels
real qLevel[15:0];

// Output
real digitalOut[3:0];

// Loop variable
integer i;

analog begin
@( initial_step ) begin

for (i = 0; i < 16; i = i +1)
begin
   qLevel[i] = vMin+i*vRef/16.0;
end
```

```
 $strobe("=============== The value of qLevel15 is %g", qLevel[15]);
 $strobe("=============== The value of qLevel3 is %g", qLevel[3]);
 $strobe("==================== The value of vMin is %g",     vMin);

end // initial_step

if ((V(vIn,AGND) >= qLevel[0]) && (V(vIn,AGND) < qLevel[1]))
begin
       digitalOut[3] = 0;  digitalOut[2] = 0;  digitalOut[1] = 0;
       digitalOut[0] = 0;
end

etc … (the rest of the if statemets)

// Assigning the digital outputs


V(D0) <+ transition(vLow * (1-digitalOut[0]) + vHigh *
(digitalOut[0]),td,tr,tf);
V(D1) <+ transition(vLow * (1-digitalOut[1]) + vHigh *
(digitalOut[1]),td,tr,tf);
V(D2) <+ transition(vLow * (1-digitalOut[2]) + vHigh *
(digitalOut[2]),td,tr,tf);
V(D3) <+ transition(vLow * (1-digitalOut[3]) + vHigh *
(digitalOut[3]),td,tr,tf);

end // analog
endmodule
```

5.4.3 Combining Quantizer and ADC

Before we continue to simulation we shall create the ADC too. Create a new cell in your design library, this time however, you shall start with a `schematic` view instead. Call the cell `tste16Lab1Adc`. The view is obviously very similar to a test bench.

Instantiate your sample-and-hold (`tste16Lab1SampleHold`) and your four-bit quantizer (`tste16Lab1Quantizer`) in the cell view and name them properly. Connect the output of the S&H to the input of the quantizer and name the net to something understandable. Then we need to add pins as terminals to the ADC. Press the bind key 'p' to get a pop-up window in which you can fill in the following pins:

```
vIn CLK AGND D0 D1 D2 D3
```

`vIn` and `CLK` should be input, `AGND` inputOutput and the rest should be output. The resulting output should look something like the one in figure 5.6. In order to increase readability and understanding of the block, we have also added a so called sheet to the cell view (`daisy`: `daisySheet`). Further on, you can also add text information (`shift + L`) and add comments to your schematic.

✗ It is good design practice to add history comments to your schematic - especially if you work in groups. Add for example texts like: "Added another capacitor, Sept 3, 09. /JJW" or similar.

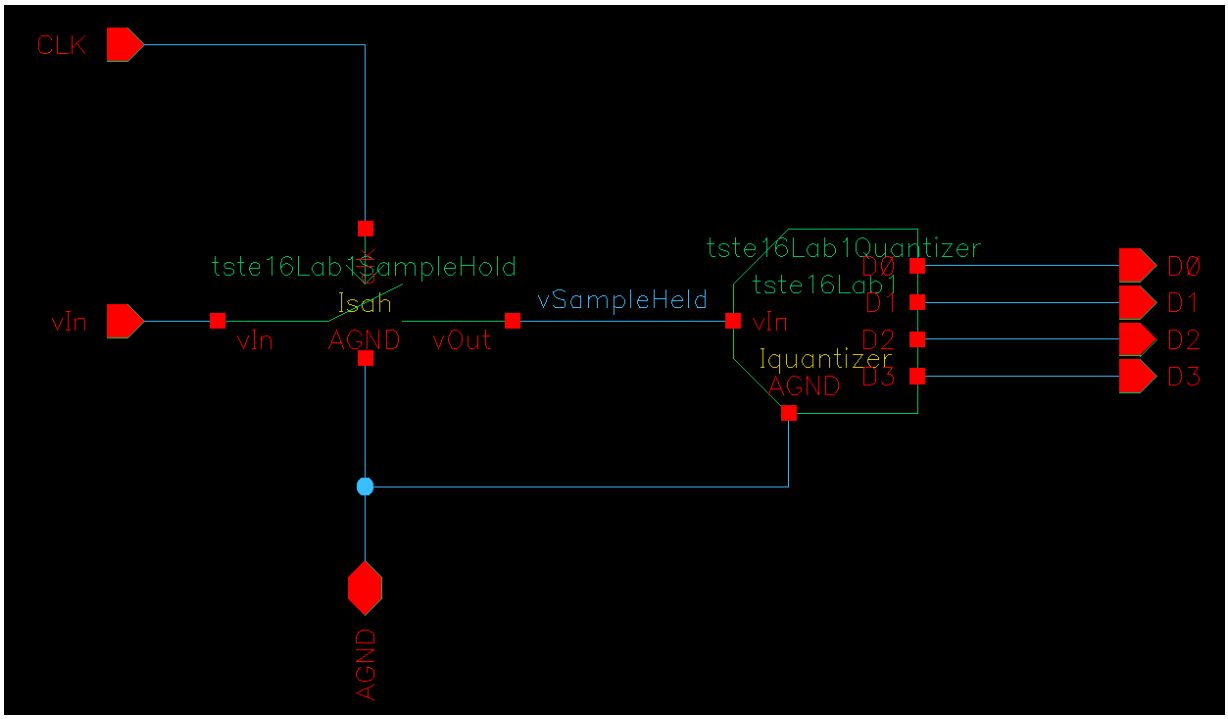

Figure 5.6: Example of the ADC using the combined S&H and quantizer.

Now also create the symbol. Use `Design > Create Cellview > From Cellview ...` from the menu system in the window. Confirm that `Tool / Data Type` is `Composer-Symbol` and then OK the form. You will now actually get a predefined symbol to work with. You can alter the pins, etc., if you want a "prettier" symbol. Check and save everything.

5.5 Setting up a Test Bench for the Whole Filter and ADC Chain

In this part we are about to simulate the system that we have designed so far. Further, the simulations performed in the Cadence environment should also be exported to matlab/octave to be able to calculate the output spectrum of the transient signals. This can actually be performed in the Cadence environment as well, but we will export the data to matlab/octave since those tools can be used for implementing the digital signal processing later on in the project. It is also used to estimate the signal-to-noise ratio (SNR) which can otherwise be difficult to do using Cadence.

The first step is to set up the test bench for the filter and ADC chain. First, create the schematic cellview `tste16Lab1Top_T` in the `tste16Lab1Test` library. Add the symbols of your components `tste16Lab1RcFilter` and `tste16Lab1Adc` from the `tste16Lab1` library. Further, include a sinusoidal voltage source (`analogLib`, `vsin`) and the analog ground (`analogLib`, `gnd` or `ctrl+shit+g`). Also include the output file writer (`daisy`, `daisyFileWriter`) which writes the simulation data to a MATLAB/octave compatible file. A clock signal is required which in this case is chosen as a pulse voltage source (`analogLib`, `vpulse`). Use the following values on the `vpulse` device.

Parameter	Value	Explanation
AC magnitude	0 V	No AC variations
AC phase	0	The phase of the AC sinusoid.
DC voltage	0	The DC voltage of the clock is zero.
Voltage 1	3.3 V	Start voltage for the pulse in the transient analysis.
Voltage 2	0 V	Second voltage for the pulse in the transient analysis.
Delay time	0	Delay from start of simulation.
Rise time	`tRiseFall`	Rise time from voltage 1 to voltage 2.
Fall time	`tRiseFall`	Fall time from voltage 2 to voltage 1.
Pulse width	`clkPeriod/2`	The width of the pulse (time it will be 0 V)
Period	`clkPeriod`	Time between repetition of the sequence.

Table 5.5: Settings for the pulse source.

The test bench should look like in Figure 5.7. Note that all nodes with the same name are connected, and thereby the clock signal (`vClk`) is connected to the `daisyFileWriter` too.

In the block `daisyFileWriter`, the output file name (use double quotes for the name: "`filename`"if you want to change the default name) and path can be specified. The digital data is then written to that file, which can be loaded into Matlab/octave for further processing and analysis.

The `daisyFileWriter` block has a 16-bit wide input bus `vIn<15:0>`. Connect your digital data outputs, {`D0 D1 D2 D3`}, to the lowest bits in this vector. The remaining bits should be connected to ground.

✗ *Cadence does not allow that multiple input signals are connected together and shorted to ground, therefore each input should be connected to ground via small resistor. This can be done using some simple index/bus notations.*
1. Insert a resistor, set the Instance Name to e.g., Ires<11:0> to instantiate 12 resistor units.
2. Connect the resistors to a gnd cell with Instance Name, e.g., Ignd<11:0>
3. You can now connect a 12-bit wide bus to the resistor. Each bus member will now have a separate ground connection to ground.

In the test bench above an analog sinusoidal input signal is filtered, sampled and quantized. The quantization is successfully modeled by the introduction of uncorrelated noise. For sinusoidal input signals, however, the signal frequency should not be chosen as a rational factor times the sampling frequency. Otherwise, the quantization error can not be modeled as uncorrelated noise. A so called coherent sampling should be used.

We will now estimate the signal-to-noise ratio (SNR) from the data written to the file. The m-file `daisyEstAdcSnr.m` (from `$WORKAREA/daisy/m`) estimates the SNR as outlined in Chapter 1.6 in the Mixed-Signal Processing Systems course book.

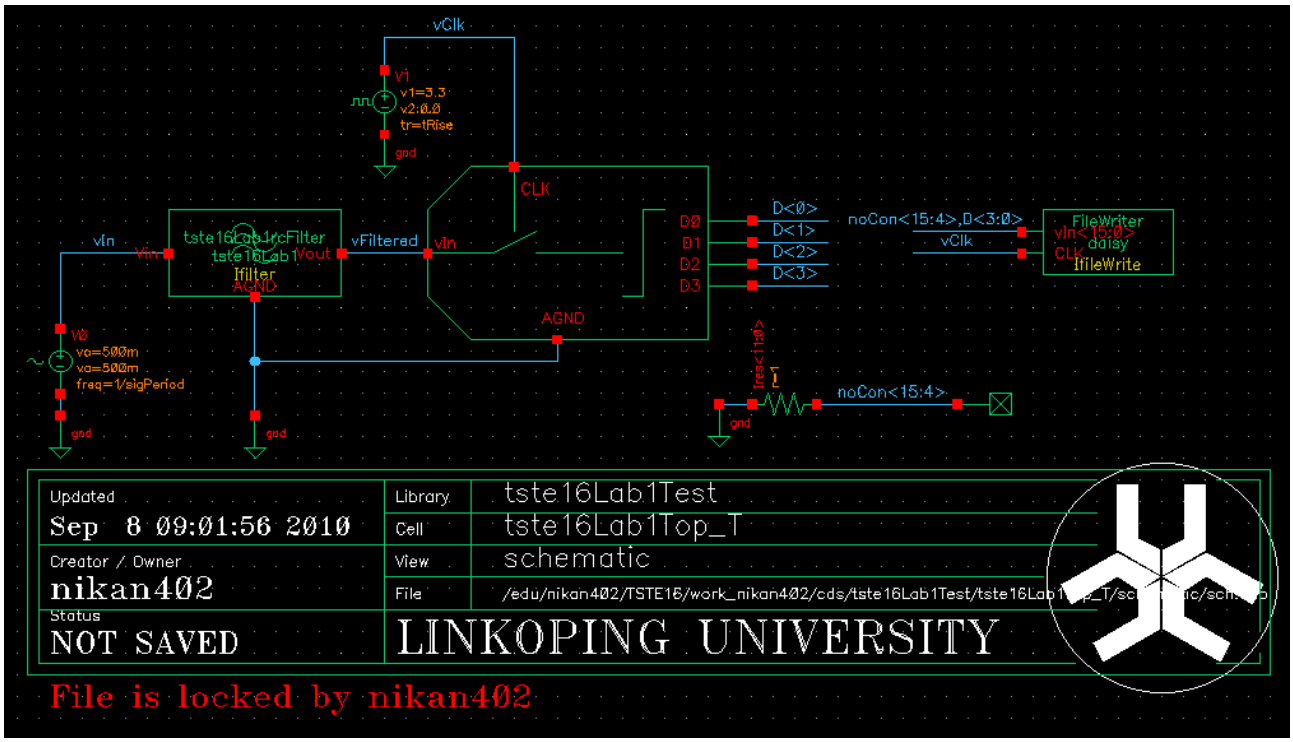

Figure 5.7: Test bench for the filter and ADC chain.

Start Matlab/octave and estimate the SNR from the simulated data. You will need to check that you apply the correct value for `wkT = 2*pi*f0/fsample`, where `f0` is the signal frequency and `fsample` is the sampling frequency. Make sure you simulate to produce at least a few hundreds of samples.

6. REFINING THE FILTER IMPLEMENTATION

A first model of the system is now complete and we are now continuing to refine the models of the individual building blocks. First we are going to start to refine the filter building block. In the first implementation only a mathematical mapping is used to realize the filtering function. However, we are now moving towards a description which is more close to a possible implementation. In this case we should implement the filtering function using an amplifier (`analogLib > vcvs`), a resistor (`analogLib > res`), and a capacitor (`analogLib > cap`). The implementation should look like the one shown in figure 6.1. Once again notice that we have added more comments (`shift + L`) and shapes (`n`) to improve the readability of our schematics for other project members to read. Components have been named properly so that we can read the netlistst easier, etc.

The amplifier is a voltage controlled voltage source (`vcvs`) and it amplifies the signal, `Vint - AGND`, `A0` times to the output between `Vout` and `AGND`. For the amplifier set the gain equal to the variable `A0`. The resistance of the resistor should be `filterR` and the capacitance of the capacitor `filterC`. Save the schematic view (`shift + y`).

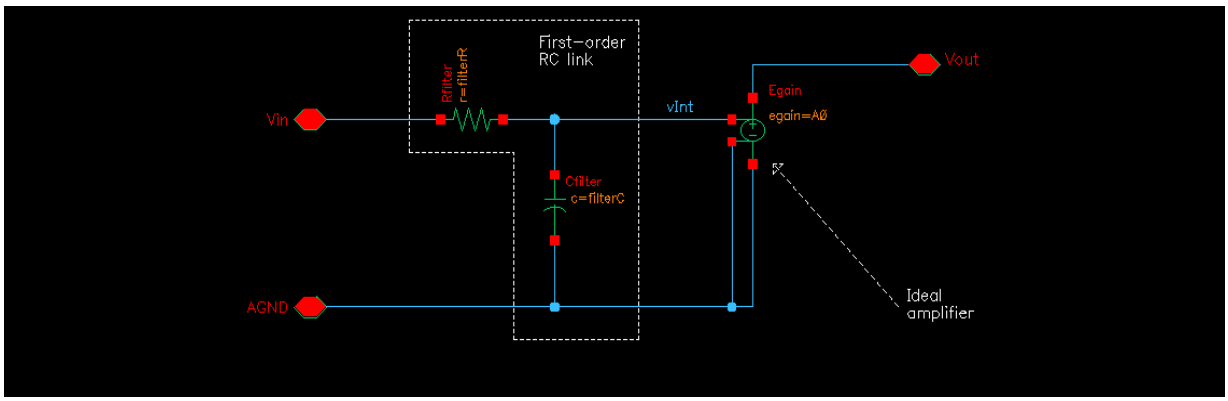

Figure 6.1: Schematic-level model of the (active) filter.

Assume that the amplifier does not load the resistor and capacitor. Compute the transfer function from the input to the internal node, `vInt`. The amplifier does not load the resistor and capacitor, but it has a gain of A0.

Compute the transfer function from `Vin` to `vInt`:	
Compute the transfer function from `Vin` to `Vout`:	
Calculate `filterR` and `filterC` so that the pole is placed at -100 kHz	

7. USING THE HIERARCHY EDITOR

Now we will start using the hierarchy editor in order to facilitate the use of different block descriptions in the filter and ADC chain. The hierarchy editor enables us to select which view we want to use for each and everyone of the different blocks in our test bench. Therefore, one can easily choose which model or level of abstraction that should be used when simulating the whole system. For example, we can simulate the ADC with a behaioral-level description and use a much more accurate description of the filtering block than the high-level model (and vice versa).

The hierarchy editor is started by creating a new cell view for the `tste16Lab1Top_T` called `config`. (Notice that the Tool field changes to `Hierarchy-Editor` if you type `config` in the `View Name` field. The hierarchy editor appears and since this now is a new cell view, we need to define the top cell of our design. In this case we verify that it says `Library: tste16Lab1`, `Cell: tste16Lab1Top_T`, and then we enter `View: schematic`. Press `Use Template` and select `Spectre`. Add `veriloga` to the `Stop List`.

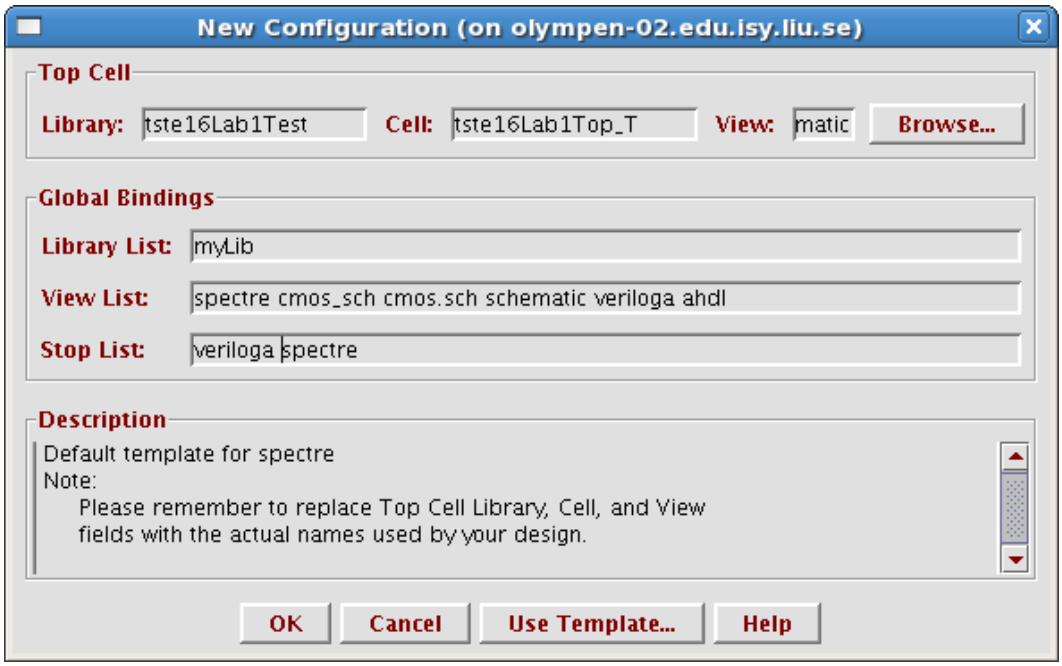

Figure 7.1: Configuration window for the hierarchy editor

Check that your window look similar to the one in Figure 7.1 before pressing OK. (Notice again, that Cadence 6.x looks a bit different, but contents in the fields are the same).

In the `Cell Bindings` you can see which cell views that are used during the simulations. For example, you can change the cell view `tste16Lab1/ tste16Lab1RcFilter/ schematic` to `tste16Lab1RcFilter/ tste16Lab1RcFilter/ veriloga` by clicking on the right mouse button on that row. A pop-up menu appears in which you should select `Set Cell View` and choose `veriloga`. After each time you have made a change update the hierarchy editor by `View > Update` or `ctrl + u`.

Save and close the hierarchy editor and go to the library manager and double click on the `config` view of `tste16Lab1Top_T`. A dialog box appears where you should tick `yes` to both questions. Both the hierarchy editor and the top cell view schematic windows are now displayed.

The simulation can be performed in the same way as before, eventhough you are using the `config` cell view instead of a `schematic` cell view. Depending on the settings in the hierarchy editor, different descriptions of the building blocks will now be used and netlisted by the simulator.

7.1 Simulating the Filter and ADC Chain With Different Filter Descriptions

We are now about to compare the simulation results for the two different implementations of the RC filter. In the hierarchy editor make sure that the view for the `tste16Lab1RcFilter` still is `veriloga`. Set the value of the first pole of the filter equal to 100 kHz. Simulate the circuit using the ADE and check the transient signal at the output of the `tste16Lab1RcFilter`. Make sure to simulate some periods of the input signal.

We should now compare this simulation with a simulation using the schematic description of the RC filter. First, we need to store the waveform for the current simulation. This is done by the menu alternative `Window > Subwindows` in the Waveform window. Select the transient response and press the `Disable Results Update` button. This analysis will not be updated for the coming simulations until this window is closed. Hence, only minimize this window.

Change the cell view used for simulation for the `tste16Lab1RcFilter` to schematic in the hierarchy editor (right mouse button, etc.), update the definitions (`ctrl + u`) and save the hierarchy description. Copy variables for the test bench into the simulator window and add appropriate values to the `filterR` and `filterC` variables and set `A0` to unity. (If the variables do not appear in the ADE check that you have changed the view, updated, and saved in the hierarchy editor. Sometimes one can run a dummy simulation to get the variables into the window.). Run the simulation and make sure that the pole for this implementation has been placed at (-)100 kHz. Compare the two simulation (you can change the color of the waveforms by double clicking on the waveforms and changing the color property. Use drag-and-drop for moving one of the curves on top of the other.).

Comments on the simulated results

8. WRAP-UP

In this laboratory we have used the Cadence design environment to design a filter and ADC chain. We have also incorporated a top-down design methodology where we start to describe our circuits using a model and further on refining the model towards a real implementation using physical components. Here, we have used two descriptions of the filter implementation and only one for the ADC (S&H/Quantizer). However, typically several levels of abstraction are used. The continuation of the laboratory for a real chip implementation would for the filter part be to realize the unity-gain amplifier, using e.g. an operational amplifier. This amplifier can also be described by a number of different models, etc. The two last models for the implementation of a real integrated circuit would be a transistor-level implementation of the amplifier and its corresponding layout level description.

Coming back to the laboratory design task, we can see that your implementation can be viewed as shown in Figure. For example, the first model of the filter and ADC chain is in this case the mathematical descriptions shown in the second row of the diagramme. This mathematical description of the filter can be replaced by the model consisting of an ideal resistor, capacitor, and amplifier illustrated by the third row in the diagramme. By iteratively simulating both model one and two (and onwards), the impact of the model refinement can be seen and understood. This procedure has been illustrated by Figure 8.1. This is one of the advantages with the top-down design procedure.

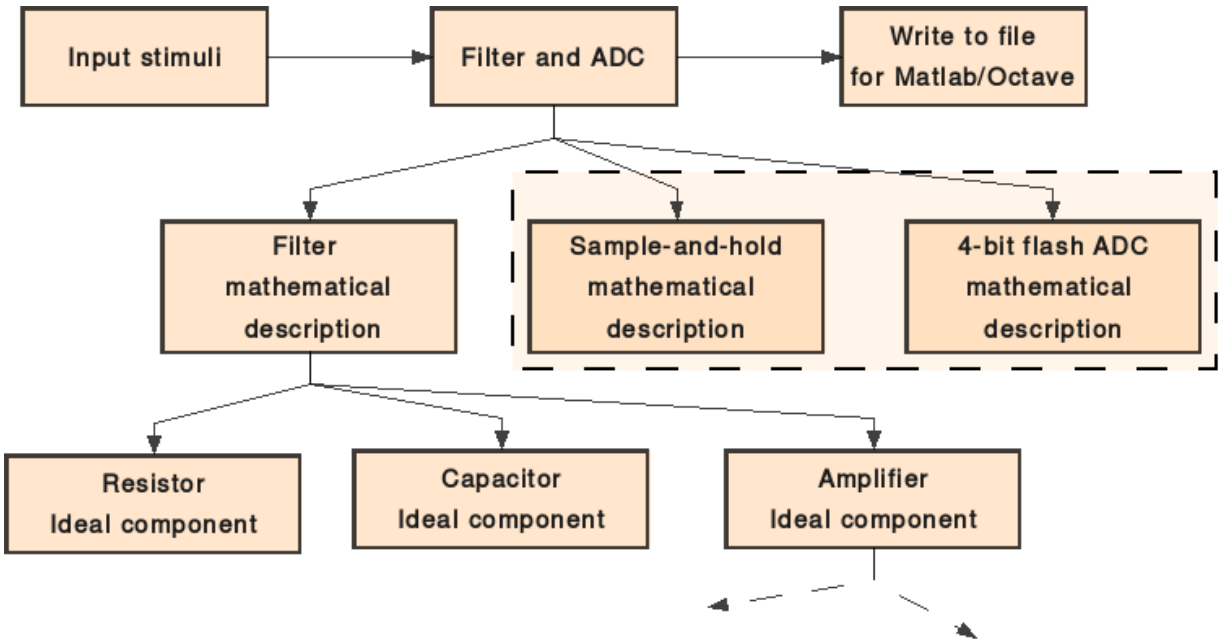

Figure 8.1: The top-down description of the filter and ADC chain. All descriptions in the leaves can be further refined as illustrated by the dashed lines for the amplifier.