Qucs

Work Book

Thierry Scordilis
Mike Brinson
Gunther Kraut
Stefan Jahn
Chris Pitcher

Contents

Introduction

Important note and warning

You should take into account the fact that this document is written on the fly, so some mistakes are still possible, and the author is not responsible for any damage due to the use of this document.

This document is intended to be a work book for RF and microwave designers. Our intention is not to provide an RF course, but some touchy RF topics. The goal is to insist on some design rules and work flow for RF desings using CAD programs. This work flow will be handled through different chapters on quite different subjects.

Work book content

In this workbook, we will pass through some regular tasks. But there is a progression on the explanations, and due to the fact that we have to cover a huge amount of information, some key point will be shown ony once, so it is recommanded to read the chapters in order.

This work book will include:

Work flow: the regular process of project design is shown,

Understanding RF data sheets: a usual task, that could be hell, could turn a project into a nightmare,

BJT Modeling: after having chosen a device, we always need to use in the CAD, and usually this device does not exits in the CAD... how to create it and verify

DC static: since all active devices have to be biased...

PA Desgin: the active component is found, and a small amplifier is designed without to many constraints

LNA Design: a more constraint design using more rules, stability, noise etc.

oscillator design: a procedure that is typical from CAD issues, handling non usual procedure,

vco design: a normal evolution from a oscillator,

detector: a design difficult to handle.

more will come . . .

1 General Design Flow

Knowing the fact that you are familiar with the regular design flow of RF, microwave circuits and or systems, we need to clarify how *Qucs* is intended to be used for this type of circuits design.

As an RF research engineer, I'm still having some new graduate students. And I'm always having some problems with the new methods that are teached. Usually they arrive with some knowledge on CAD programs, but they do not really know how to dimension their design. They use only the optimizer to replace their thinking. What a pity! Of course not all of them are like this, but it is a common trend. By since work book I want to show that there are some rules to follow, and that a design can be calculated, and that it will not work due to a wizard!

For the experts, nothing very new herein, but only some particular use of Qucs, since the design rules are the one that you could have on the workbench using a paper and a pen.

The author.

Regular document organisation

We will try to have always the same kind of organization inside the different chapters, that is to say:

a main topic: in order to say in which field of activity this design is intended to be used

a block specification: in order to know what we have to do. This task will not be explain at a first glance, since it is not the goal of this document (we're not dealing with system specification, it could be if the component present in Qucs are increased ...so why not in further version of this document.)

DC explanation: if the design includes a DC part, then we should provide the DC study including thermal aspect if needed.

functional design: in order to explain how this functionality is designed either in general or by the mean of Qucs. The second aspect should be always kept in mind. Everything might not be straightforward on other CAD programs, and therefore not considerated herein.

Hoping that these explanations clarifies the goal of this document.

2 Getting started with Qucs

2.1 Introduction

The following sections are meant to give an overview about what the Qucs software can be used for and how it is used to achieve this.

Qucs is free software licensed under the General Public License (GPL). It can be downloaded from `http://qucs.sourceforge.net` and comes with the complete source code. Every user of the program is allowed and called upon (on a voluntary basis of course) to modify it for their purposes as long as changes are made public. Contact the authors to verify them and finally to incorporate it into the software.

The software is available for a variety of operating systems including

- GNU/Linux
- Windows
- FreeBSD
- MacOS
- NetBSD
- Solaris

On the homepage you'll find the source code to build and install the software. Build instructions are given. Also links for binary packages for certain distributions (e.g. Debian, SuSE, Fedora) can be found.

Once the software has been successfully installed on your system you can start it by issuing the

```
# qucs
```

command or by clicking the appropriate icon on your start menu or desktop. Qucs is a multi-lingual program. So depending on your system's language settings the Qucs graphical user interface (GUI) appears in different languages.

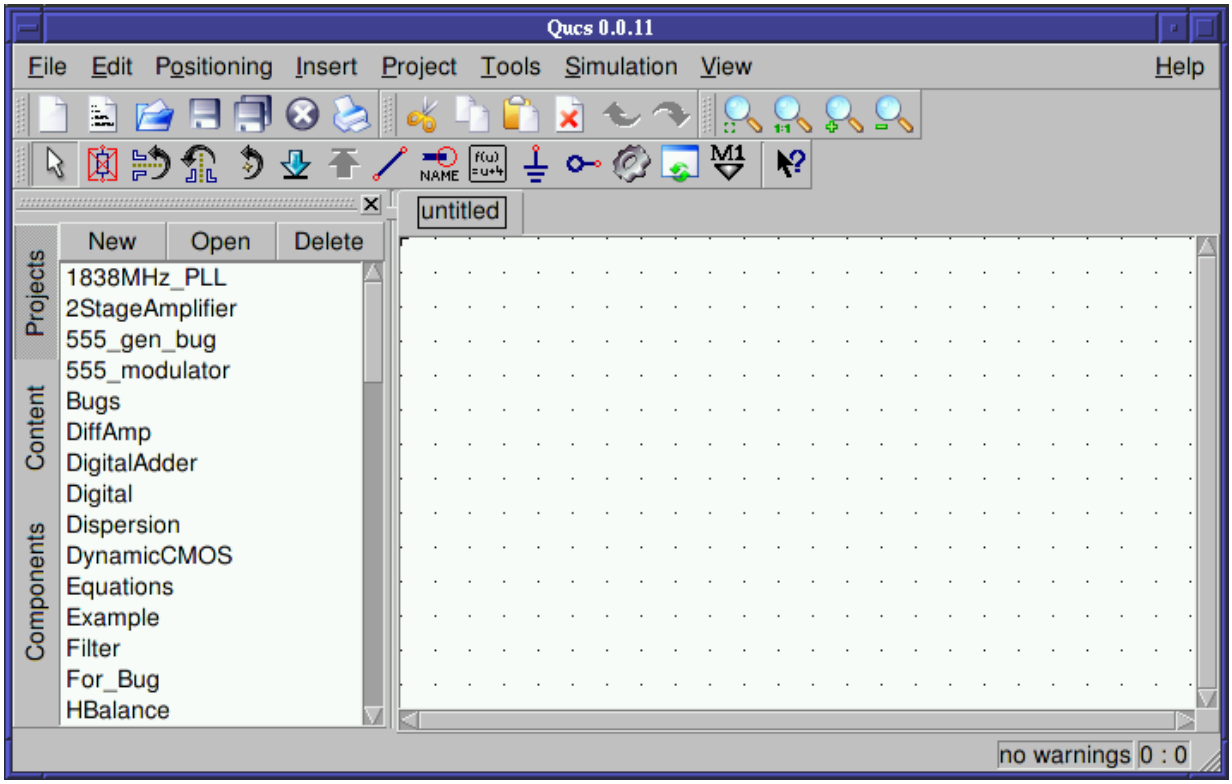


Figure 2.1: Qucs has been started

On the left hand side you find the **Projects** folder opened. Usually the projects folder will be empty if you use Qucs for the first time. The large area on the right hand side is the schematic area. Above you can find the menu bar and the toolbars.

In the **File** → **Application Settings** menu the user can configure the language and appearance of Qucs.

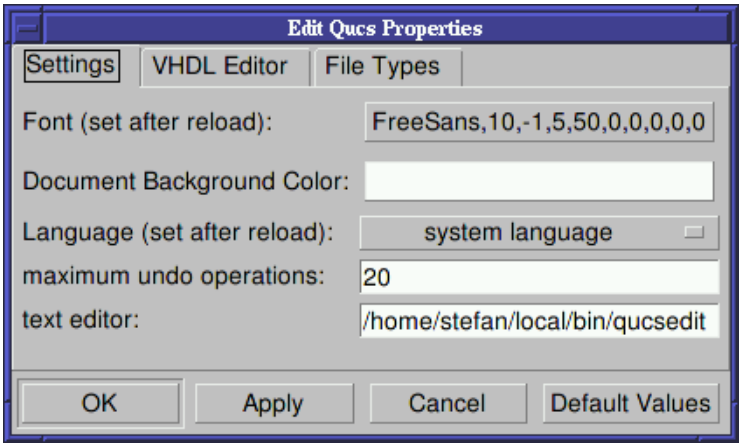


Figure 2.2: Application setting dialog

To take effect of the language and font settings the application must be closed either via the Ctrl + Q shortcut or the **File** → **Exit** menu entry. Then start Qucs again.

2.2 Tool suite

Qucs consists of several standalone programs interacting with each other through the GUI. There are

- the GUI itself,

  The GUI is used to create schematics, setup simulations, display simulation results, writing VHDL code, etc.

- the backend analogue simulator,

  The analogue simulator is a command line program which is run by the GUI in order to simulate the schematic which you previously setup. It takes a netlist, checks it for errors, performs the required simulation actions and finally produces a dataset.

- a simple text editor,

  The text editor is used to display netlists and simulation logging informations, also to edit files included by certain components (e.g. SPICE netlists, or Touchstone files).

- a filter synthesis application,

  The program can be used to design various types of filters.

- a transmission line calculator,

  The transmission line calculator can be used to design and analyze different types of transmission lines (e.g. microstrips, coaxial cables).

- a component library,

  The component library manager holds models for real life devices (e.g. transistors, diodes, bridges, opamps). It can be extended by the user.

- an attenuator synthesis application,

  The program can be used to design various types of passive attenuators.

- a command line conversion program

  The conversion tool is used by the GUI to import and export datasets, netlists and schematics from and to other CAD/EDA software. The supported file formats as well as usage information can be found on the manpage of **qucsconv**.

Additionally the GUI steers other EDA tools. For digital simulations (via VHDL) the program FreeHDL (see `http://www.freehdl.seul.org`) is used. And for circuit optimizations ASCO (see `http://asco.sourceforge.net`) is configured and run.

2.3 Setting up schematics

The following sections will enable the user to setup some simple schematics. For this we first create a new project named "WorkBook". Either press the **New** button above the projects folder or use the menu entry **Project** → **New Project** and enter the new project name.

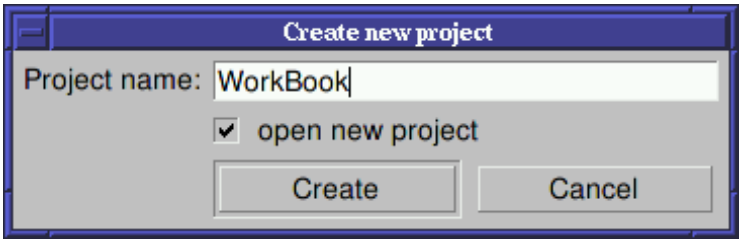


Figure 2.3: New project dialog

Confirm the dialog by pressing the "Create" button. When done, the project is opened and Qucs switches to the **Content** tab.

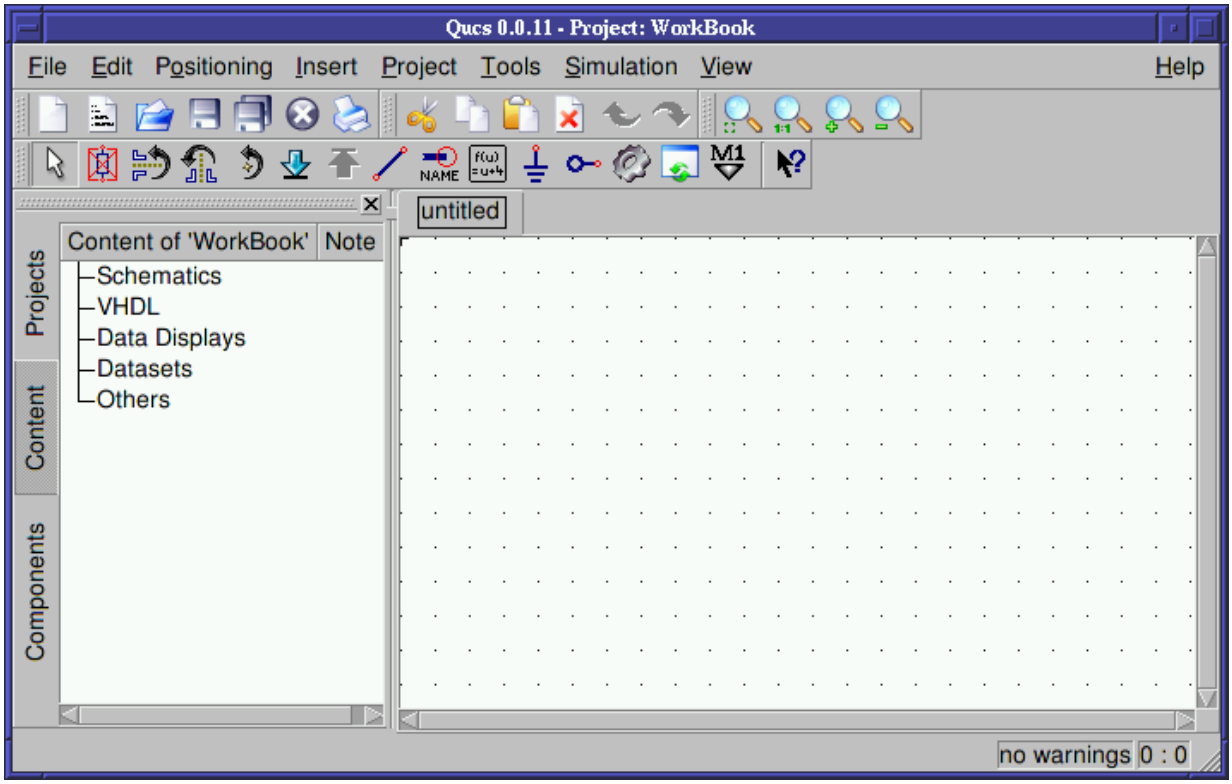


Figure 2.4: New empty project has been created

In the **Content** tab you will find all data related to the project. It contains your schematics, the VHDL files, data display pages, datasets as well as any other data (e.g. datasheets). On the right hand side an "untitled" and empty schematic window is displayed.

Now you can start to edit the schematic. The available components can be found in the **Components** tab.

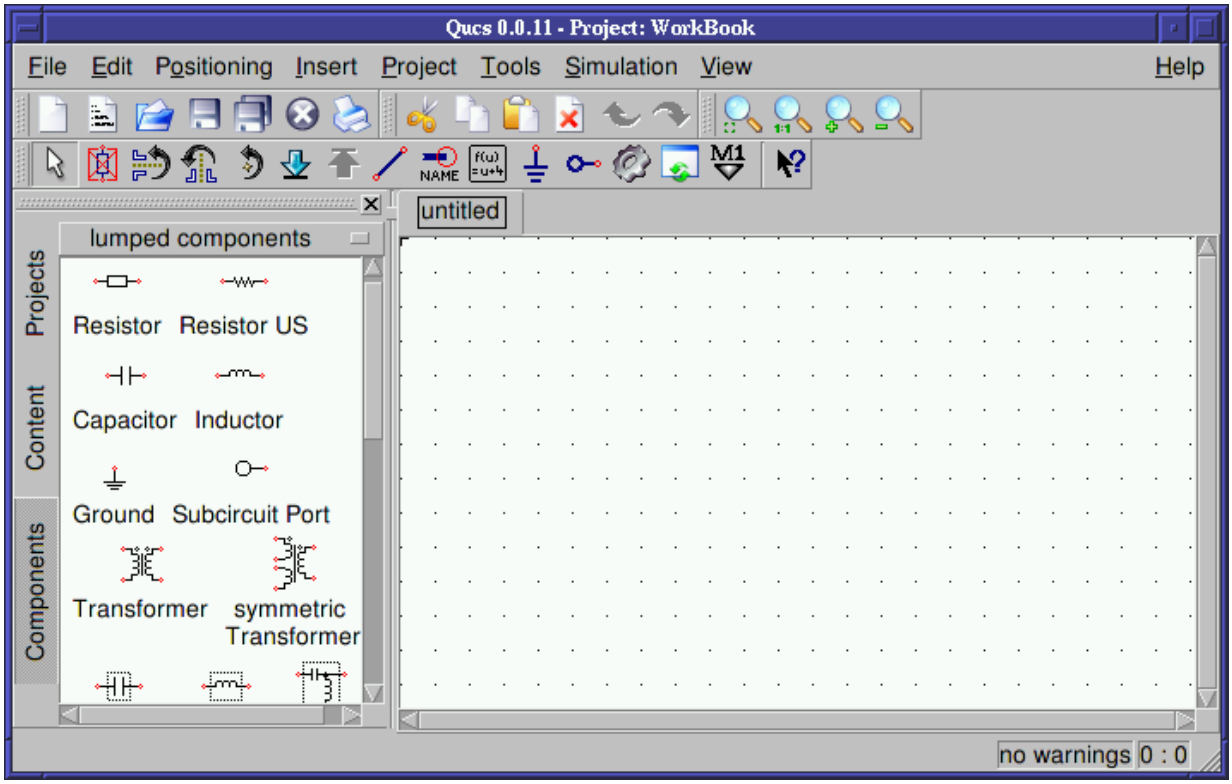


Figure 2.5: Components tab

In fig. 2.5 is shown when clicking the **Components** tab. There are lumped components (e.g. resistors, capacitors), sources (e.g. DC and AC sources), transmission lines (e.g. microstrip, coaxial cable, twisted pair), nonlinear components (e.g. ideal opamp, transistors), digital components (e.g. flip-flops), file components (e.g. Touchstone files, SPICE files), simulations (e.g. AC or DC analysis), diagrams (e.g. cartesian or polar plot) and paintings (e.g. texts, arrows, circles).

Each of the components can placed on the schematic by clicking it once, then move the mouse cursor onto the schematic and click again to put it on its final position. During the mouse move you can right click in order to rotate the component into its final position. The user can also drag-and-drop the components.

2.3.1 DC simulation - A voltage divider

The DC analysis is a steady state analysis. It computes the node voltage as well as branch currents of the complete circuit. The given circuit in fig. 2.6 is going to divide the voltage of a DC voltage source according to the resistor ratio.

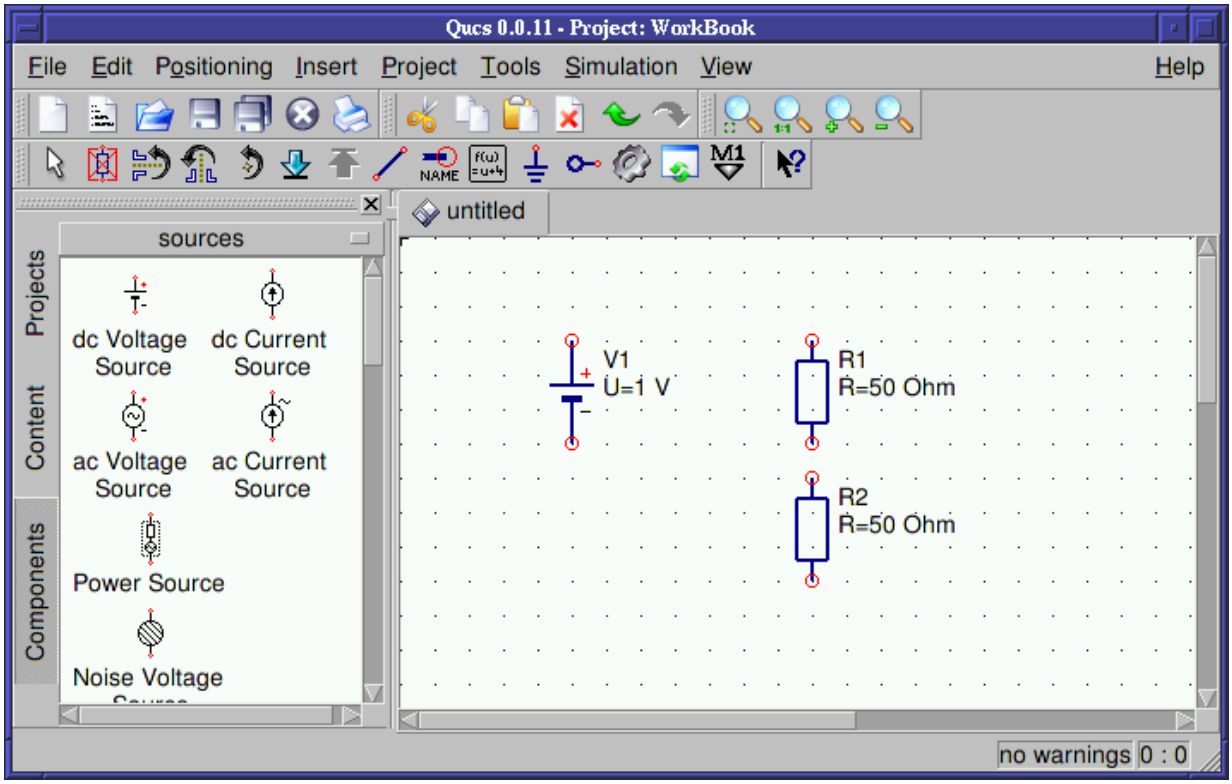


Figure 2.6: Components of the voltage divider place in the schematic area

Wiring components

Now you need to connect the components appropriately. This is done using the wiring tool. You enable the wiring mode either by clicking the wire icon or by pressing the Ctrl + E shortcut. Left clicking on the components' ports (small red circles) starts a wire, clicking on a second port finishes the wire. In order to change the orientation of the wire right click it. You can leave the wiring mode by the pressing Esc key.

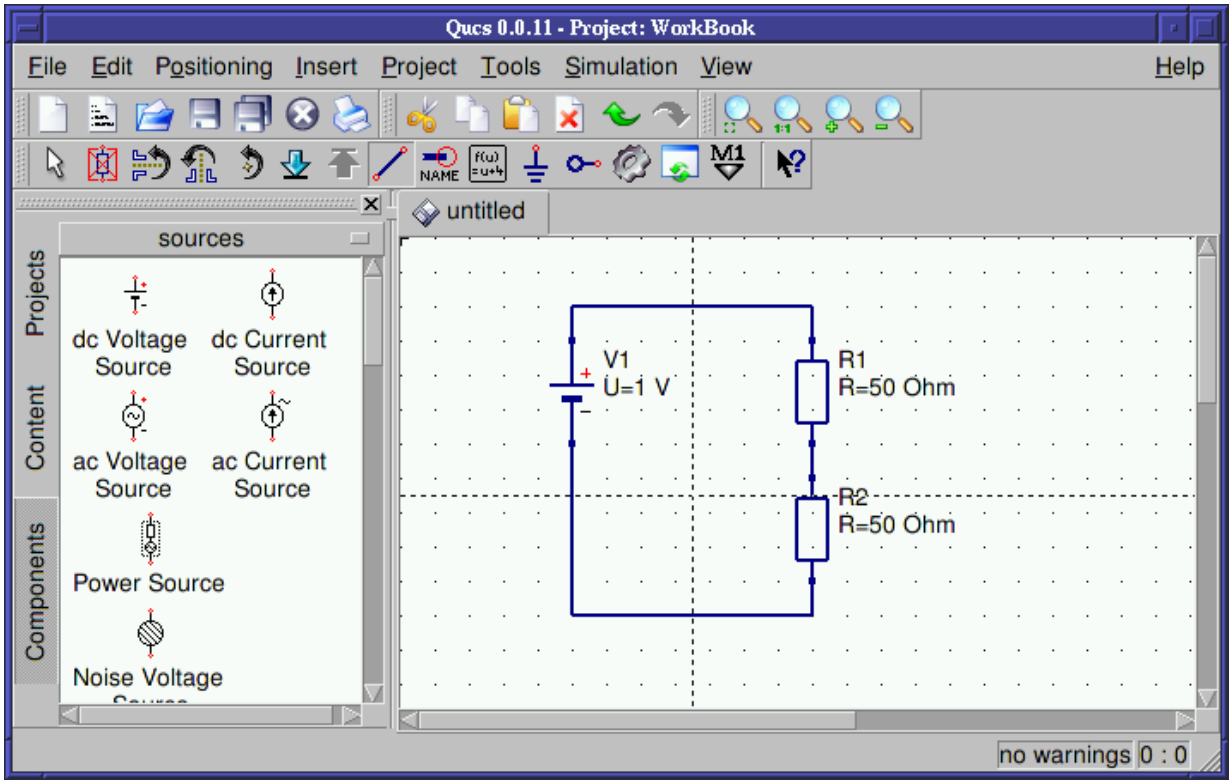


Figure 2.7: Components of the voltage divider appropriately wired

For any analogue simulation (including the DC simulation) there is a reference potential required (for the nodal analysis). The ground symbol can be found in the **Components** tab in the **lumped components** category. The user can also choose the ground symbol icon or simply press the Ctrl + G shortcut. In the given circuit in fig. 2.8 the ground symbol is placed at the negative terminal of the DC voltage source.

Placing simulation blocks

The type of simulation which is performed must also be placed on the schematic. You choose the "DC simulation" block which can be found in the **Components** tab in the **simulations** category.

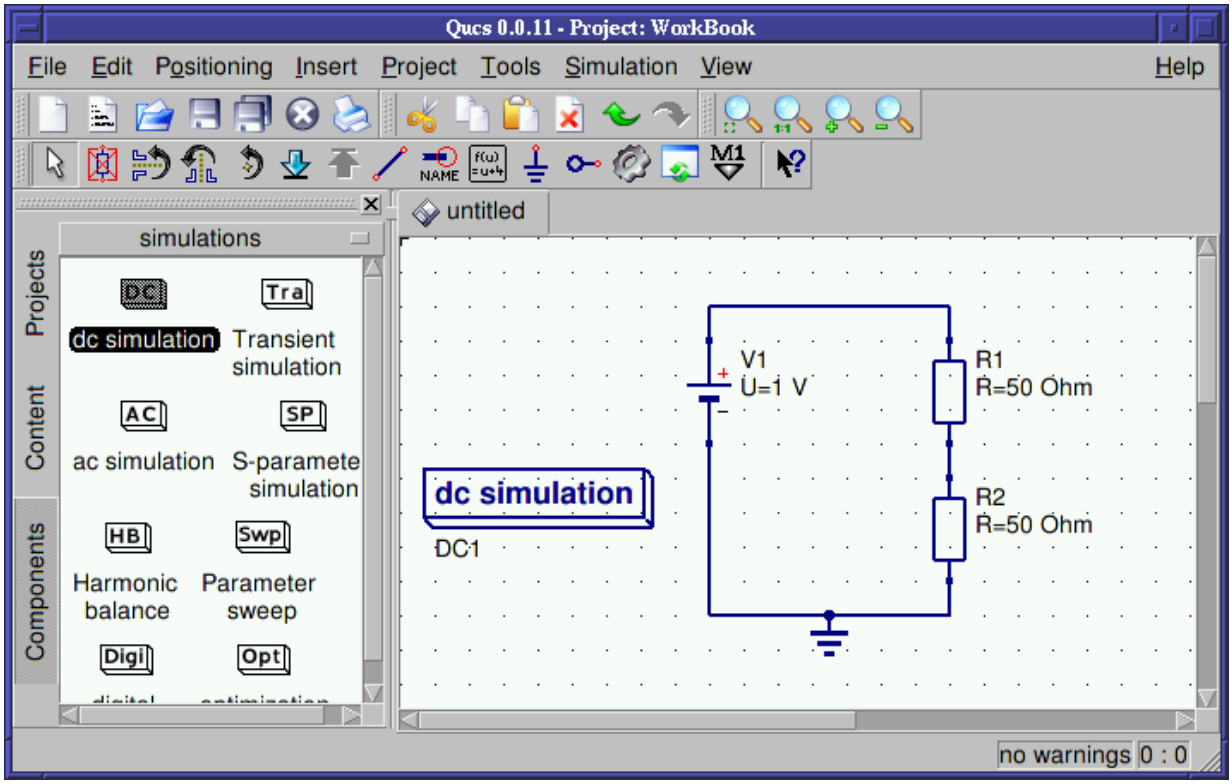


Figure 2.8: Ground symbol as well as DC simulation in place

Labelling wires

If you want the voltage between the two resistors (the divided voltage) be output in the dataset after simulation the user need to label the wire. This is done by double clicking the wire and given an appropriate name. Wire labelling can also be issued using the icon in the toolbar, by pressing the Ctrl + L shortcut or by choosing the **Insert** → **Wire Label** menu entry.

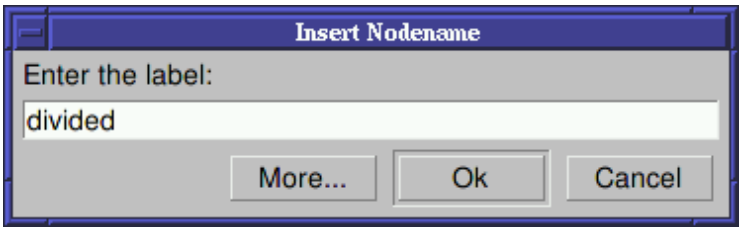


Figure 2.9: Node label dialog

The dialog is ended by pressing the Enter key of pressing the "Ok" button.

Now the complete schematic for the voltage divider is ready and can be saved. This can by achieved by choosing the **File** → **Save** menu entry, clicking the single disk icon or by

pressing the Ctrl + S shortcut.

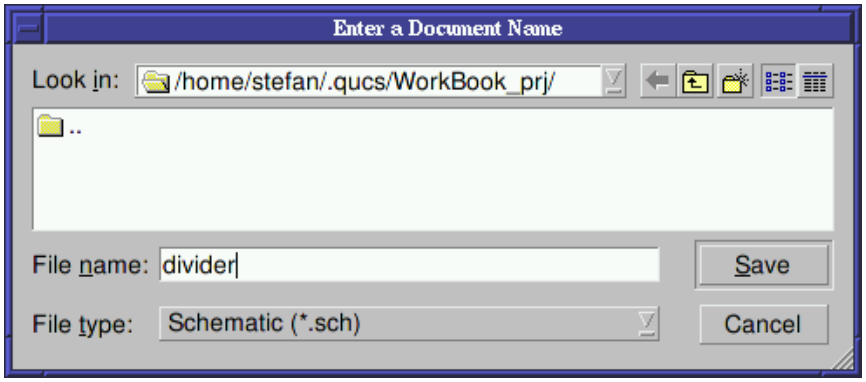


Figure 2.10: File save dialog

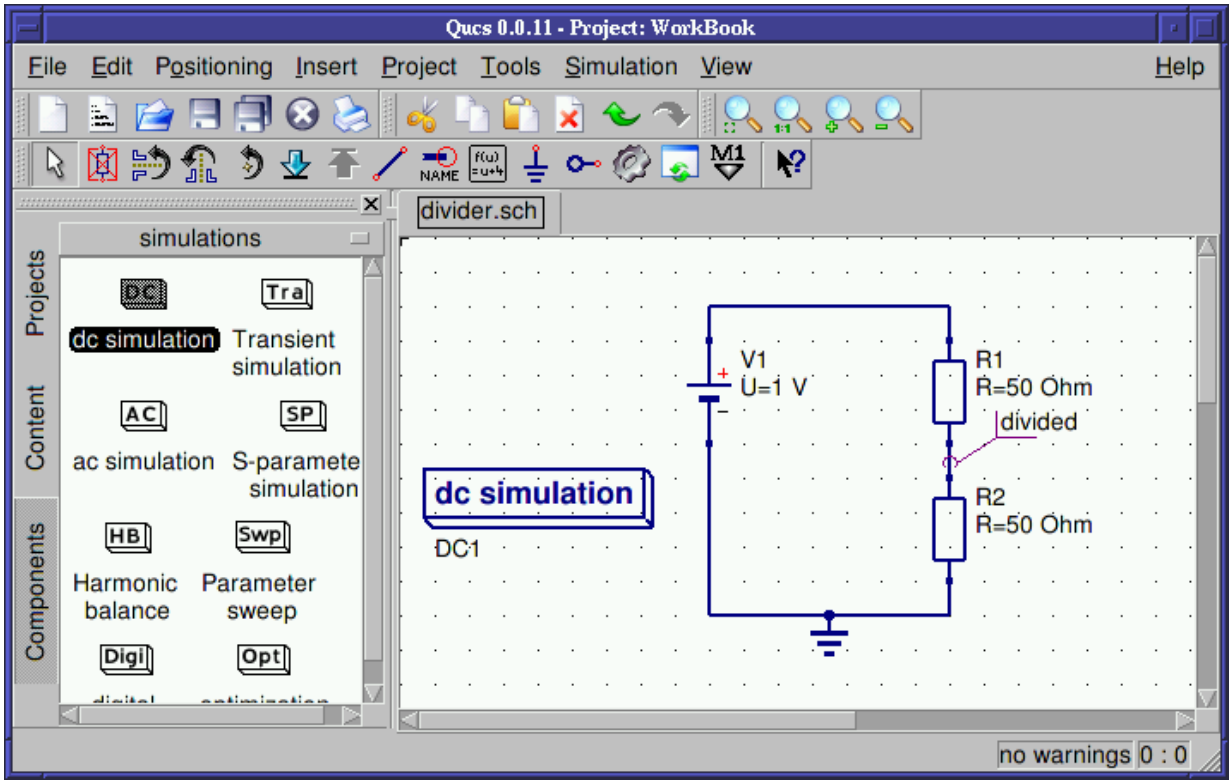


Figure 2.11: Final voltage divider schematic

The final DC voltage divider is shown in fig. 2.11.

Issuing a simulation

The schematic can now be simulated. This is started by choosing the **Simulation** → **Simulate** menu entry, clicking the simulation button (the gearwheel) or by pressing the F2 shortcut.

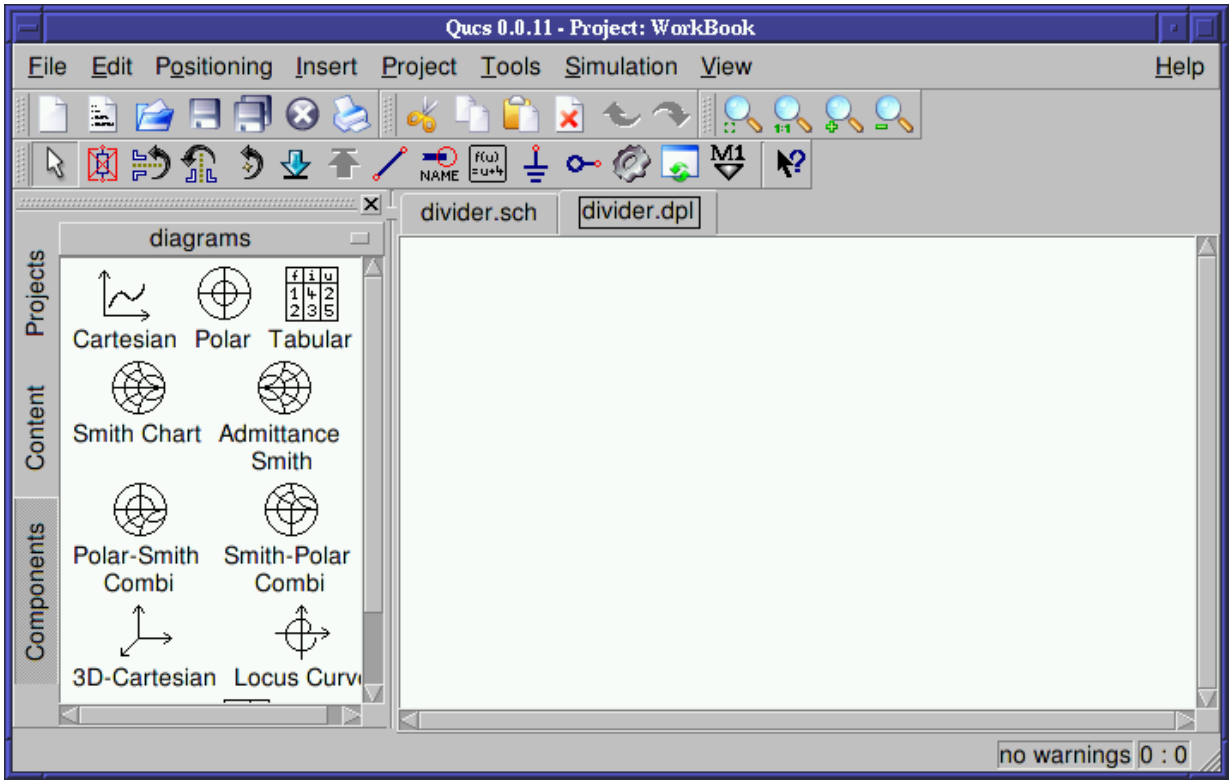


Figure 2.12: Empty data display after simulation finished

After the simulation has been finished the related data display is shown (see fig.2.12). Also the **Components** tab has changed its category to “diagrams”.

Placing diagrams

Choose the tabular (list of values) diagram and place it on the data display page. After dropping the tabular, the diagram dialog appears as shown in fig. 14.7.

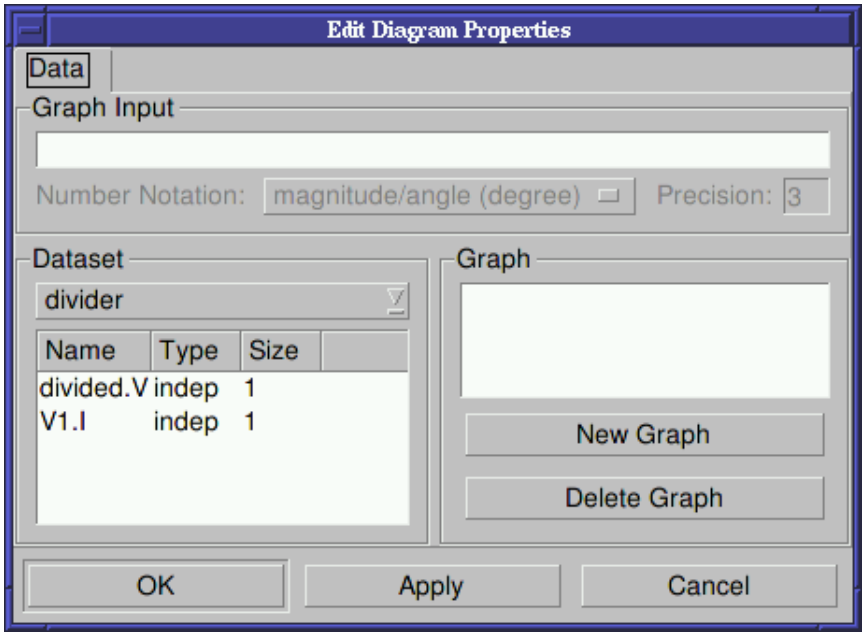


Figure 2.13: Diagram dialog

By double clicking the **divided.V** the graph (i.e. values in a tabular plot) is added to the diagram. Beside the node voltage **divided.V** also the current through the DC voltage source **V1.I** is available. Only items listed in the dataset list can be put into the graph.

Available dataset items

Depending on the type of simulation the user performed you find the following types of items in the dataset.

- *node*.**V** – DC voltage at node *node*
- *name*.**I** – DC current through component *name*
- *node*.**v** – AC voltage at node *node*
- *name*.**i** – AC current through component *name*
- *node*.**vn** – AC noise voltage at node *node*
- *name*.**in** – AC noise current through component *name*
- *node*.**Vt** – transient voltage at node *node*
- *name*.**It** – transient current through component *name*
- *S[1,1]* – S-parameter value

Please note that all voltages and currents are peak values and all noise voltages are RMS values at 1Hz bandwidth.

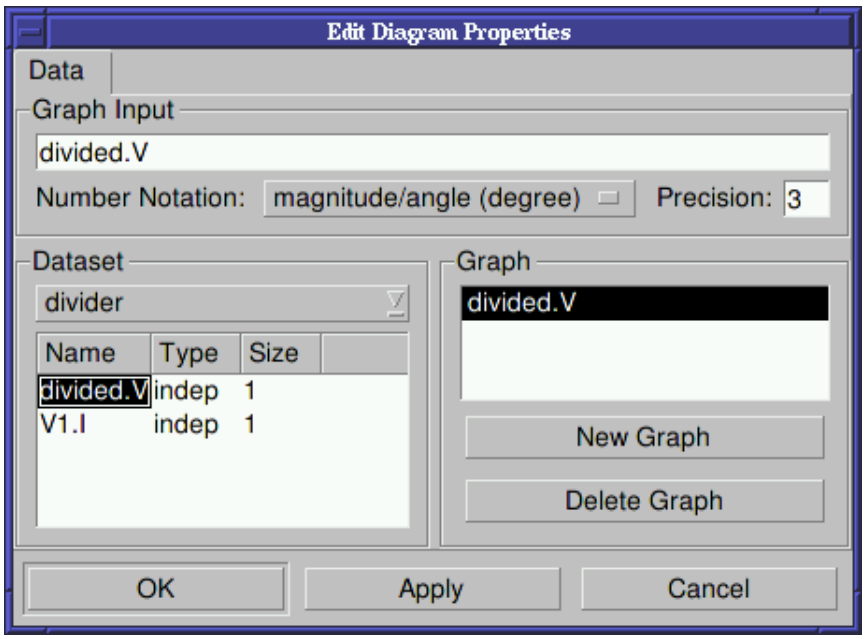


Figure 2.14: Diagram dialog with the node voltage added

Depending on the type of graph you have various options to choose for the graph. For a tabular graph there is the the number precision as well as type of number notation (important for complex values). Press the “Ok” button to close the dialog.

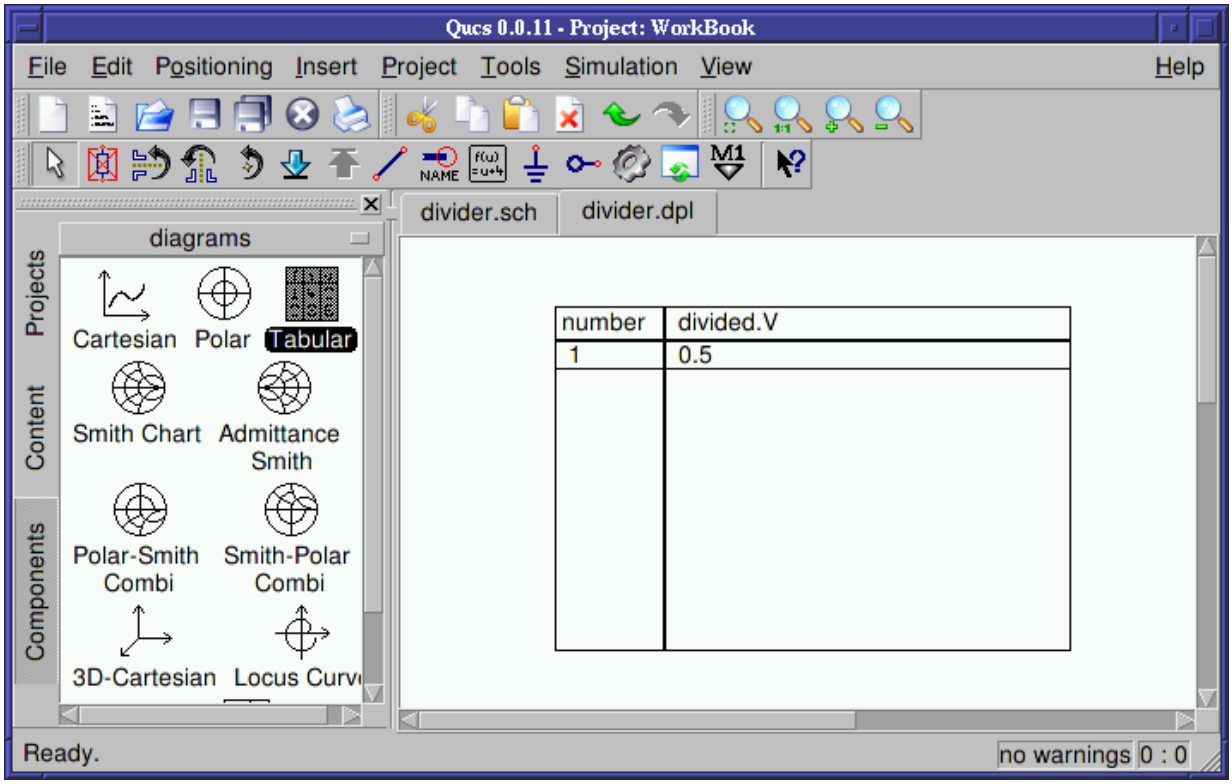


Figure 2.15: Data display with tabular graph

In the tabular graph you see now the value of the node voltage **divided.V** which is 0.5V. That was expected since the values of the resistors are equally sized and the DC voltage source produces 1V.

Congratulations! You made your first successful simulation using Qucs.

Changing component properties

If you want to change the resistor ratio then switch back to your schematic either by clicking on the **divider.sch** tab, by pressing the F4 shortcut or by choosing the **Simulation** → **View Data Display/Schematic** menu entry. Afterwards double click the **R1** resistor. This opens the component property dialog shown in fig. 2.16.

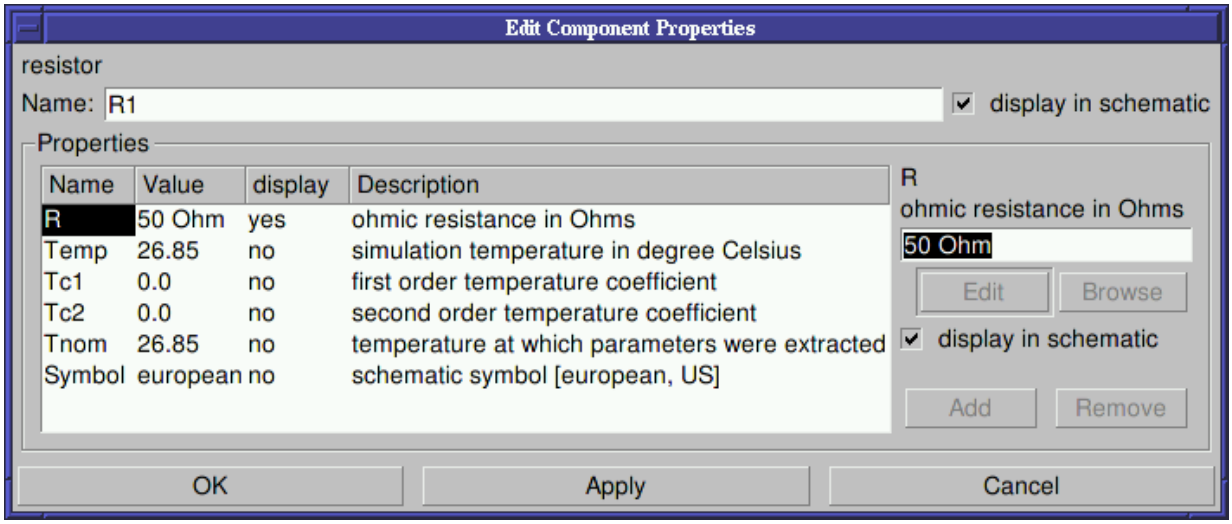


Figure 2.16: Component property dialog for the R1 resistor

In the component property dialog all the properties of a given component can be edited. A short description is given as well as there is a checkbox for each property **display in schematic** which can be used to add the property name and value on the schematic (or to hide it).

Allowed property values For component values either standard (1000), scientific (1e-3) or an engineering (1k) number notation can be chosen. Some units are also allowed. The units are

- Ohm – resistance / Ω
- s – time / Seconds
- S – conductance / Siemens
- K – temperature / Kelvin
- H – inductance / Henry
- F – capacitance / Farad
- Hz – frequency / Hertz
- V – voltage / Volt
- A – current / Ampere
- W – power / Watt
- m – length / Meter (not usable standalone, see paragraph below)

The available engineering suffixes are

- dBm – $10 \cdot \log(x/0.001)$
- dB – $10 \cdot \log(x)$
- T – 10^{12}
- G – 10^{9}
- M – 10^{6}
- k – 10^{3}
- m – 10^{-3}
- u – 10^{-6}
- n – 10^{-9}
- p – 10^{-12}
- f – 10^{-15}
- a – 10^{-18}

Please note that all units and engineering suffixes are **case sensitive** and also note the conflict in **m**. When specifying one millimeter you can use 1mm. One meter (1m) cannot be specified and will always be interpreted as one milli (engineering notation).

Now you can change the resistor value to 1Ω, see fig. 2.17.

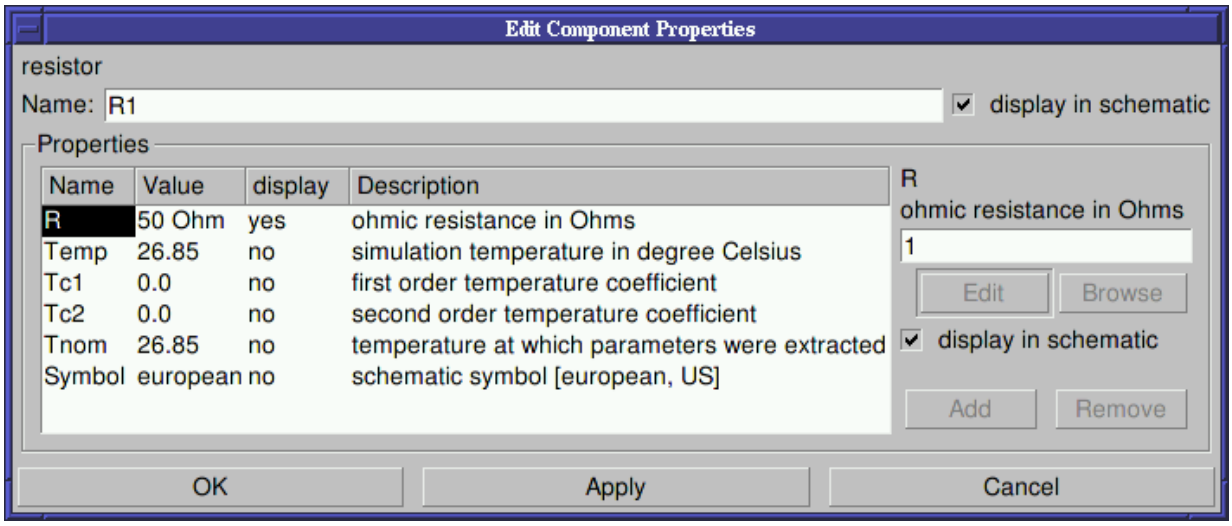


Figure 2.17: Component property dialog for the R1 resistor

Press the "OK" button to close the dialog. You will get the following schematic.

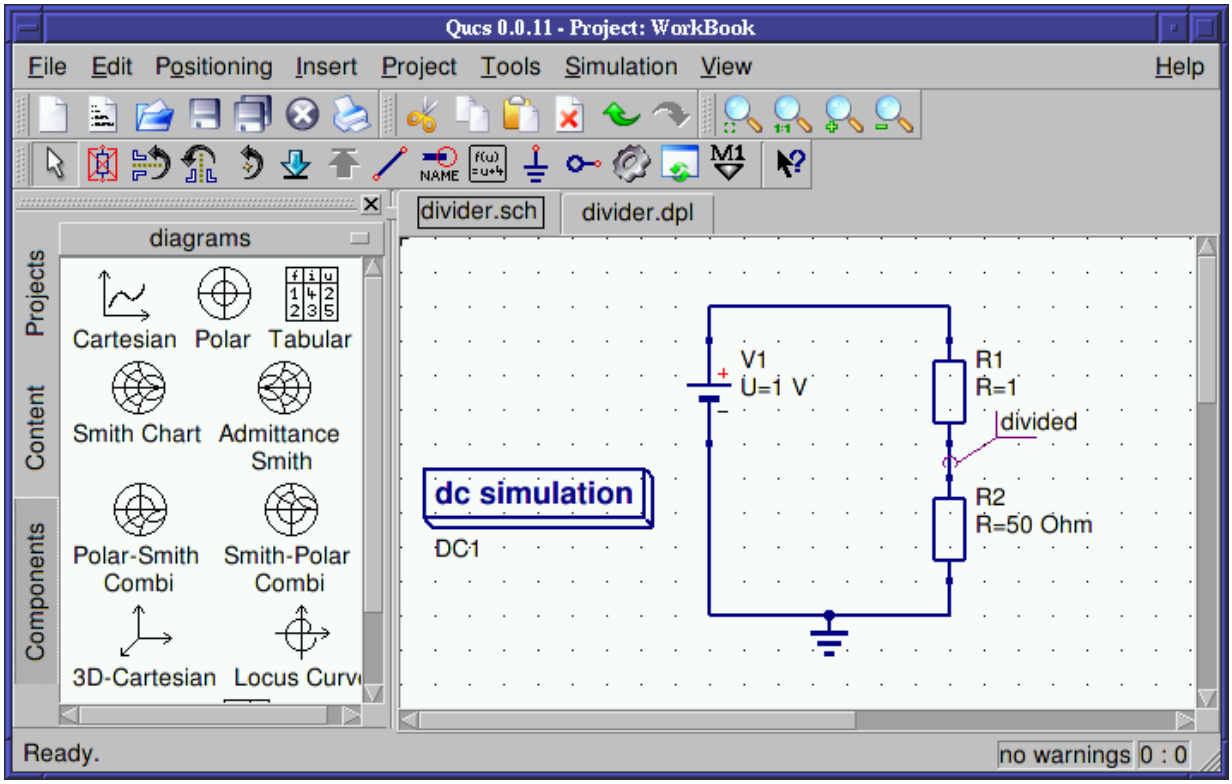


Figure 2.18: Value of resistor **R1** changed

In order to change the value of the resistor **R2** you can just click on the **50 Ohm** value directly on the schematic and edit the value.

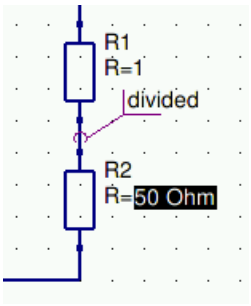


Figure 2.19: Change value of resistor **R2** directly on schematic

Change the value to "3" which will give a resistor ratio of $3/(1+3) = 0.75$. Now you have the following schematic.

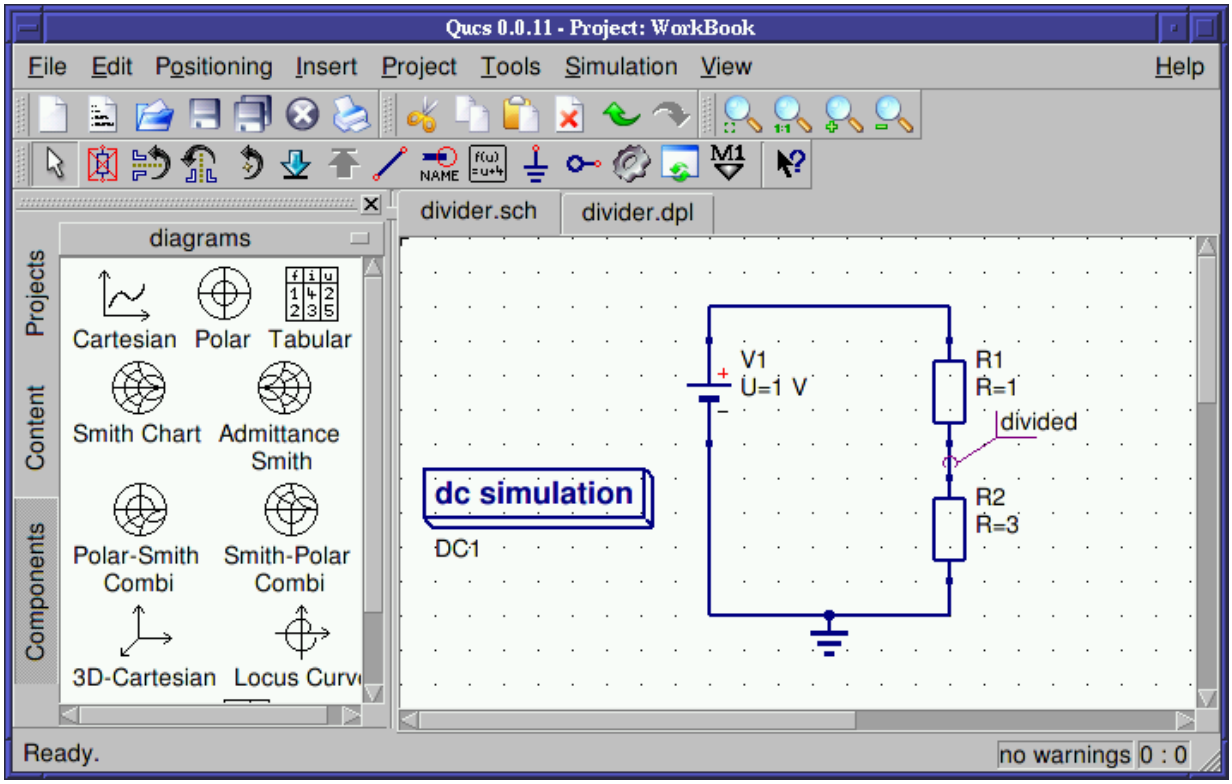


Figure 2.20: Value of resistor **R2** changed

Diagrams are not limited to be placed on the data display, they can also reside on the schematic directly. Thus you can place again now a tabular diagram on the schematic and add the **divided.V** value. The diagram will show the result from the previous simulation.

Changing document properties

If you do not want Qucs to change automatically to the associated data display you can change the behaviour in the document setting dialog. You can go to the document settings dialog by right clicking on free space on the schematic area and choose the **Document Settings** menu item in the context menu which pops up or by choosing the **File** → **Document Settings** menu entry.

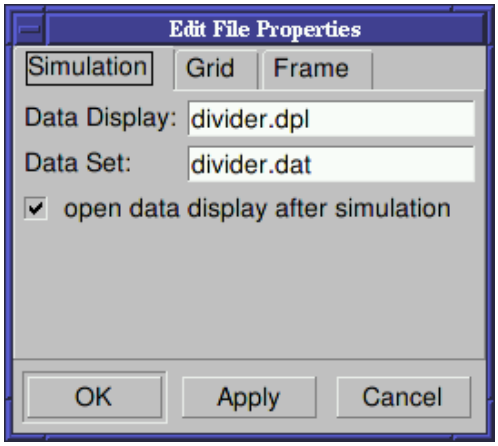


Figure 2.21: Document settings dialog

In the dialog you uncheck the **open data display after simulation** item. Press the "OK" button to apply the change. If you now resimulate the schematic by pressing the F2 shortcut the "Qucs Simulation Messages" dialog window opens and can be left by pressing Esc . The tabular diagram now show the new value for **divided.V**.

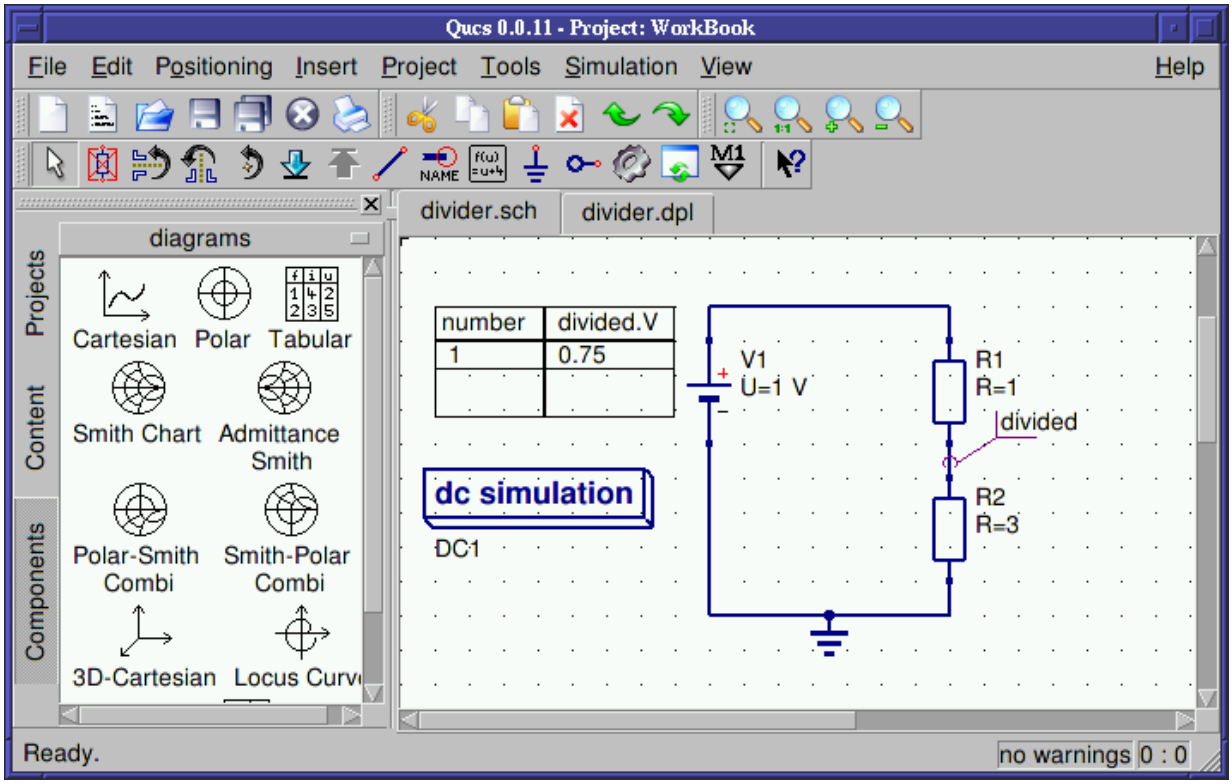


Figure 2.22: Divider schematic after new simulation

2.3.2 DC simulation - Characteristics of a transistor

We are now going ahead and will setup schematics for some characteristic curves of a bipolar transistor using DC simulation and the parameter sweep.

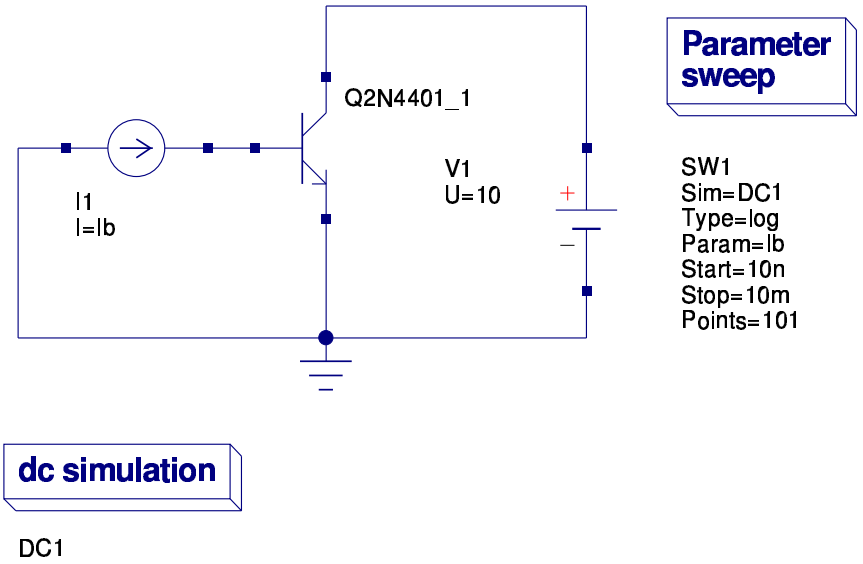


Figure 2.23: Swept DC simulation setup

In the schematic in fig. 2.23 there is a bipolar transistor placed in a common emitter configuration. Additionally a parameter sweep has been placed. Please note the **Sim** property of the parameter sweep. It contains the instance name of the DC simulation **DC1** which is going to be swept. The parameter which is swept is **Ib** (the base current) and is put into the **Param** property of the parameter sweep. The parameter **Ib** is also put into the **I** property of the DC current source **I1**.

Using the component library

The bipolar transistor has been taken from the component library. You can start the program by choosing the **Tools** → **Component Library** menu entry or by pressing the Ctrl + 4 shortcut.

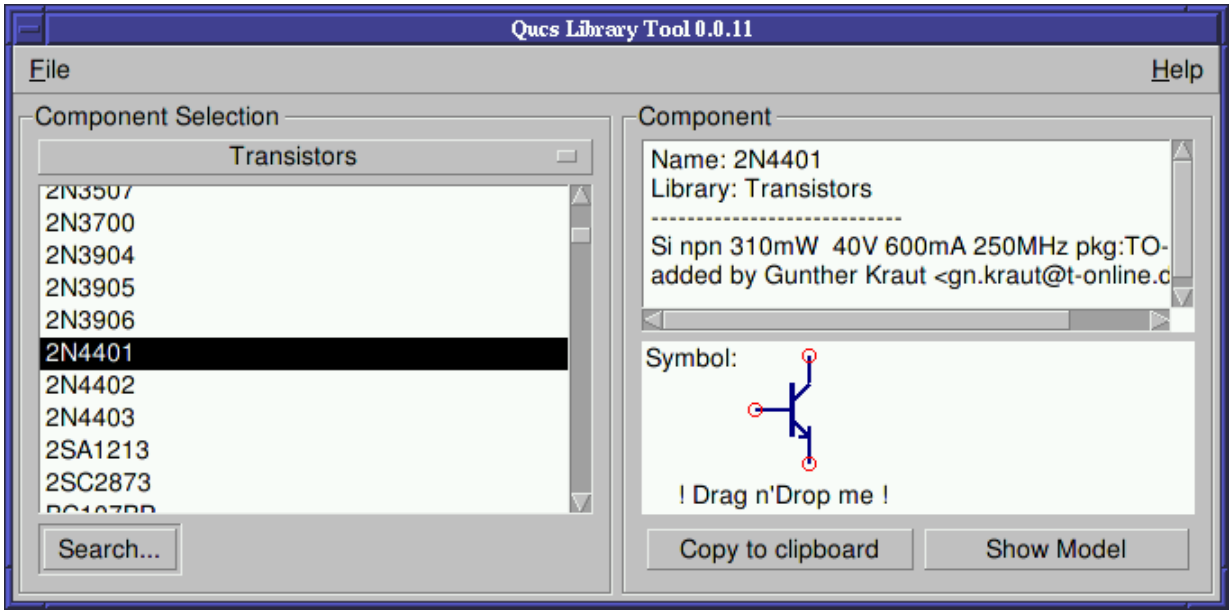


Figure 2.24: Component library tool

When choosing the "Transistor" category with the combobox you find the "2N4401" transistor. By clicking the "Copy to clipboard" button the component is available in the clipboard and can be inserted in the schematic using the Ctrl + V shortcut or by choosing the **Edit** → **Paste** menu entry. The component can also by dragged onto the schematic by clicking on the symbol in the library tool.

So what do we want to simulate actually? It is the current transfer curve of the bipolar transistor. The input current (at the base) is given by the swept parameter **Ib**. The output current (at the collector) flows through the DC voltage source **V1**. The current transfer curve is:

$$\beta_{DC} = f\left(I_C\right) = I_C / I_B$$

The current through the voltage source **V1** is the collector current flowing out of the transistor.

Placing equations on the schematic

In order to compute the necessary values for the transfer curve we need to place some equations on the schematic. This is done by clicking the equation icon or by choosing the **Insert** → **Insert Equation** menu entry. When double clicking the equation component you can edit the equations to be computed.

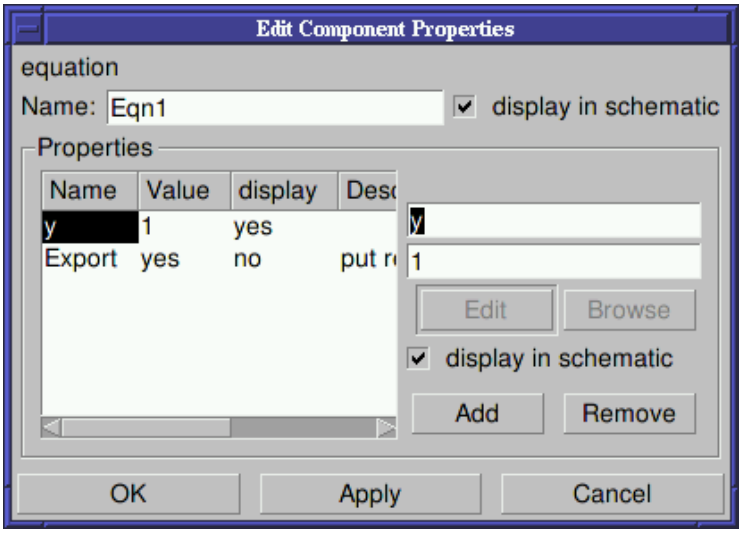


Figure 2.25: Equation dialog

In the upper edit box you enter the name of the equation and in the lower one the computation formula. The resulting schematic is shown in fig. 2.26.

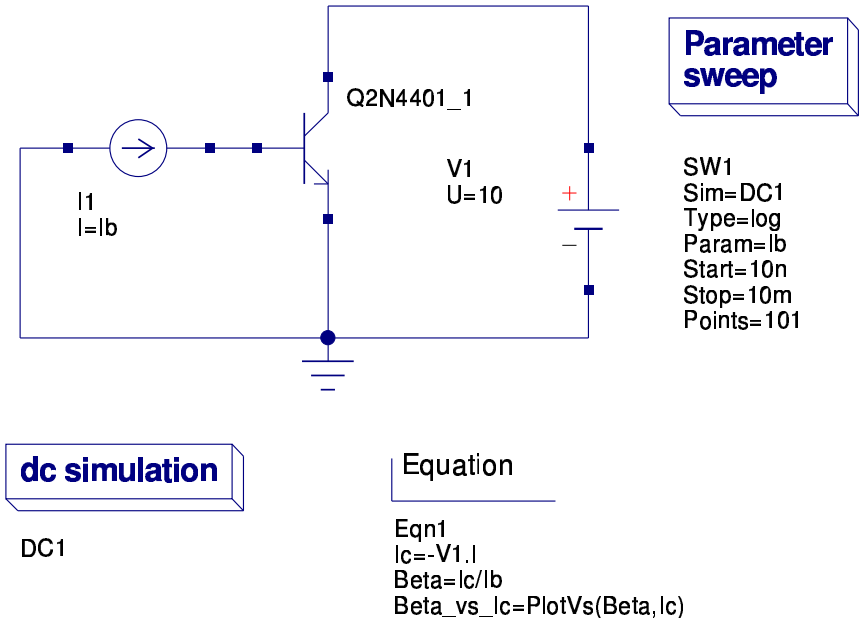


Figure 2.26: Swept DC simulation setup with equations

Note that three equations have been added. The first one **Ic=-V1.I** is the collector current flowing into the transistor (current though voltage sources flow from the positive terminal to the negative terminal). The equation **Beta=Ic/Ib** computes the current gain and finally **Beta_vs_Ic=PlotVs(Beta,Ic)** changes the data dependency of the current gain to be the collector current. The original data dependency is the swept parameter **Ib**.

The internal help system

The full list of available functions in the equation solver can be seen in the internal help system. It is started by pressing the F1 shortcut or by choosing the **Help → Help Index** menu entry. In the sidebar choose the "Short Description of mathematical Functions" entry.

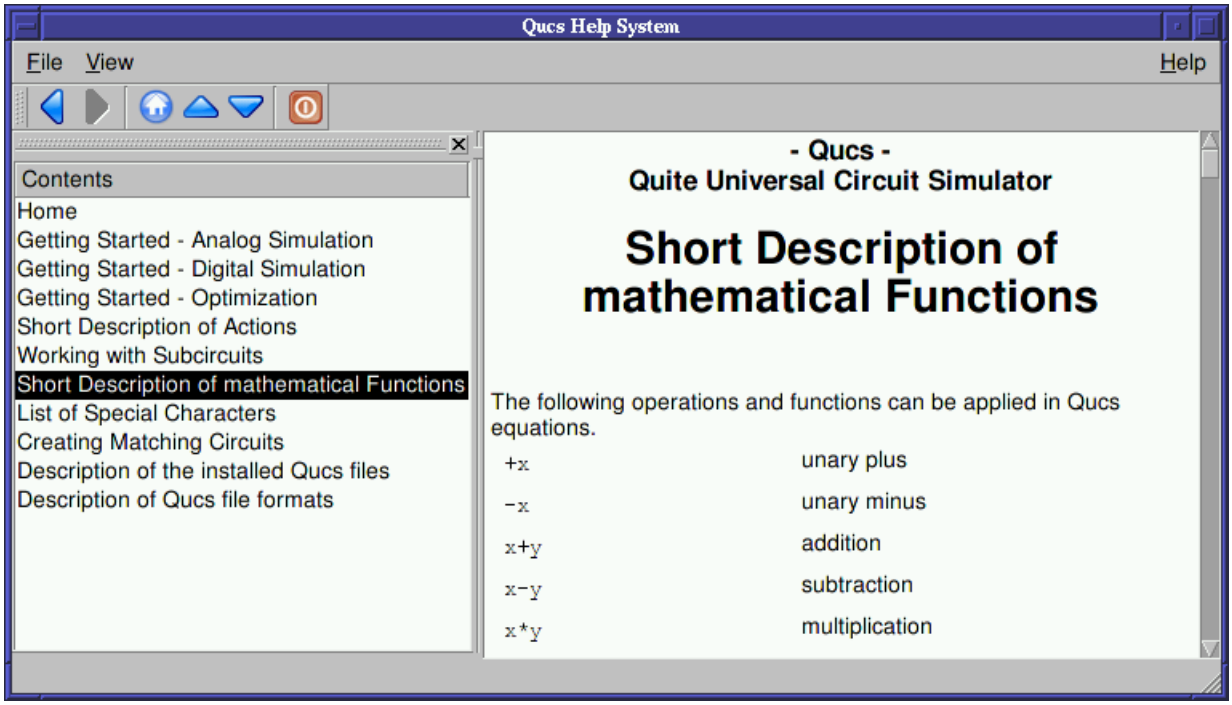


Figure 2.27: Internal help system

The help can be closed using the Ctrl + Q shortcut.

Configuring cartesian diagrams

In fig. 2.28 the final simulation result is shown. In the diagram dialog the **Beta_vs_Ic** dataset entry was chosen.

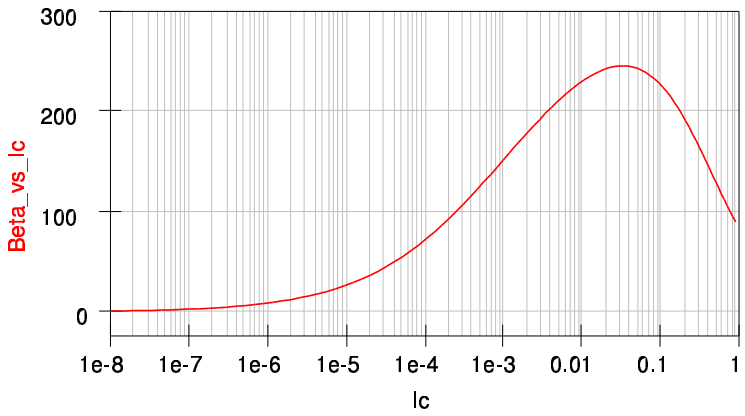


Figure 2.28: Simulation result

Additionally the x-axis has been chosen to be logarithmic. The x-axis label is **Ic**.

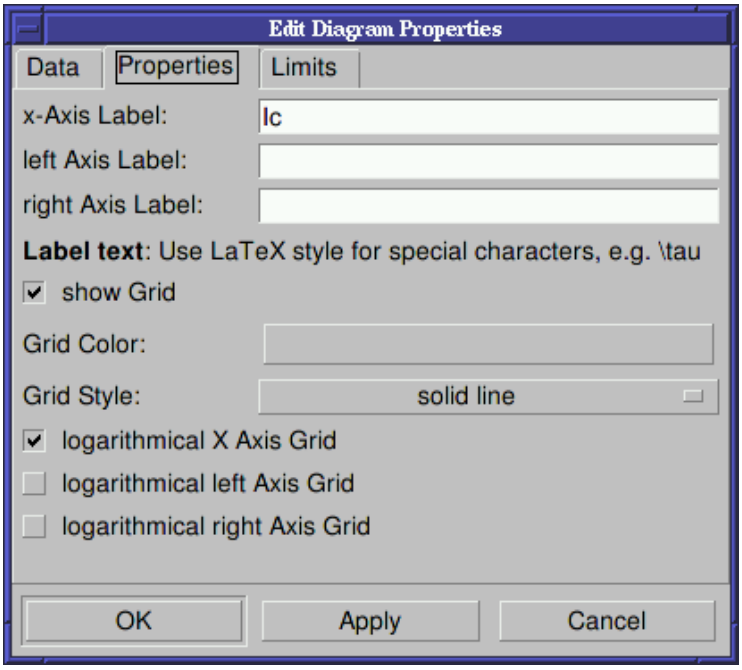


Figure 2.29: Editing diagram properties

Working with markers in diagrams

The current gain curve in diagram in fig. 2.28 shows a maximum value. If you want to know the appropriate values it is possible to use markers for this purpose.

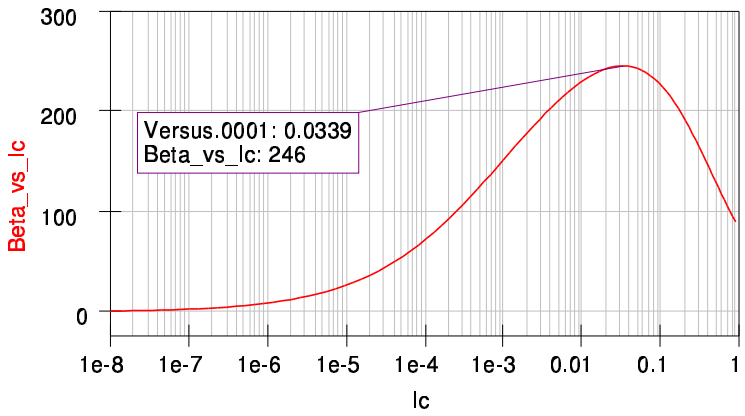


Figure 2.30: Cartesian diagram with marker

This is achieved by pressing the Ctrl + B shortcut, clicking the marker icon or choosing the **Insert → Set Marker on Graph** menu entry. Then click on the diagrams curve you want to have the marker at. If the marker is selected you can move it by pressing the arrow keys ←, →, ↑ and ↓ for multi-dimensional graphs.

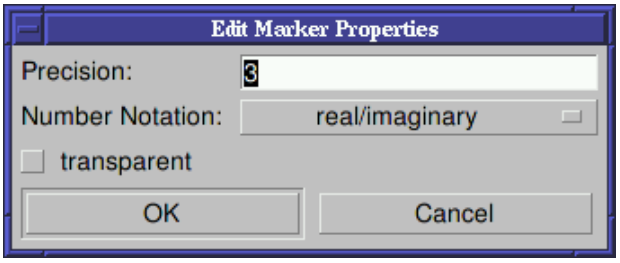


Figure 2.31: Marker dialog

Double clicking the marker opens the marker dialog. There you can configure the precision as well as the number notation of the displayed values.

A multi-dimensional sweep

Now we are going to create a schematic for the output characteristics of the bipolar transistor. The characteristic curve is defined as follows:

$$I_C = f\left(I_B, V_{CE}\right)$$

Thus it is necessary to modify the schematic from the previous sections a bit.

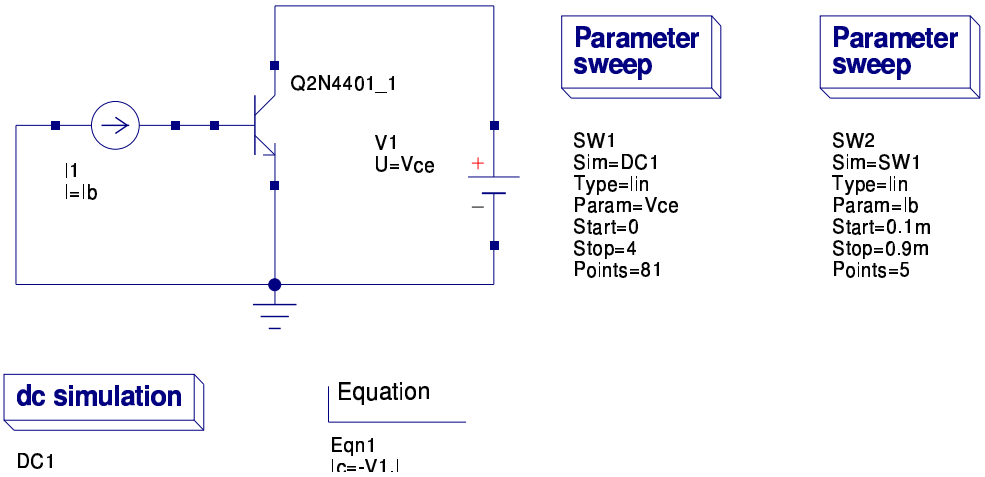


Figure 2.32: Sweep setup for the output characteristics

A second parameter sweep has been added. The first order sweep is **Vce** specified in the parameter sweep **SW1**. The **Sim** parameter points to the instance name of the DC simulation **DC1**. The second order sweep is **Ib** specified in the parameter sweep **SW2**. The **Sim** parameter of this second sweep points to the instance name of the first sweep **SW1**. The first order sweep variable **Vce** is put into the **U** property of the DC voltage source **V1**.

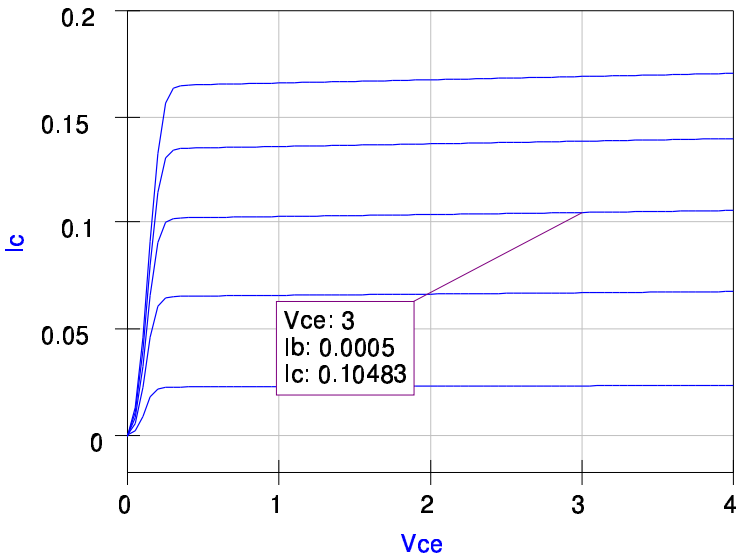


Figure 2.33: Output characteristics of a NPN bipolar transistor

2.3.3 AC simulation - Transit frequency of a bipolar transistor

In the next section we are going to determine the transit frequency of the bipolar transistor used in the previous DC sections. First a bias point is chosen. In fig. 2.34 the DC setup was a bit modified.

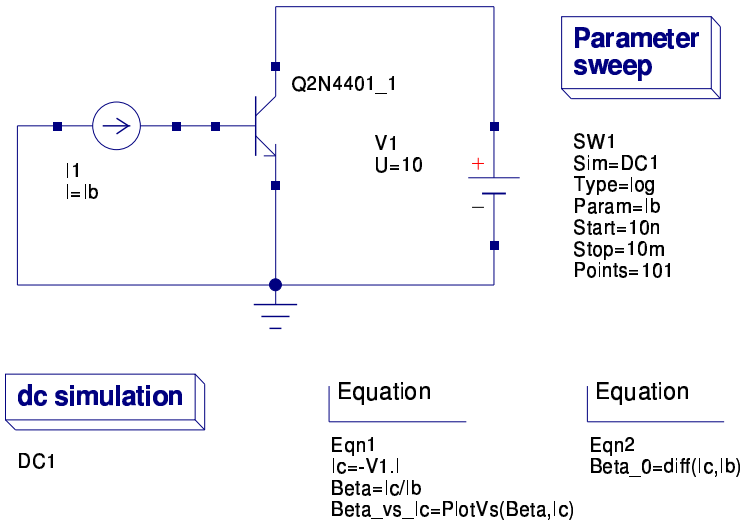


Figure 2.34: DC setup for determining a bias point for AC simulation

There is now an additional equation computing the RF current gain for zero frequency which is **Beta_0=diff(Ic,Ib)**. The equation denotes

$$\beta_{RF}\,(f=0) = \frac{\partial Ic}{\partial Ib}$$

In fig. 2.35 the DC current gain from fig. 2.30 is plotted versus the base current **Ib** choosing **Beta** in the diagram dialog instead of **Beta_vs_Ic**. The appropriate base current shown in the marker is 140µA.

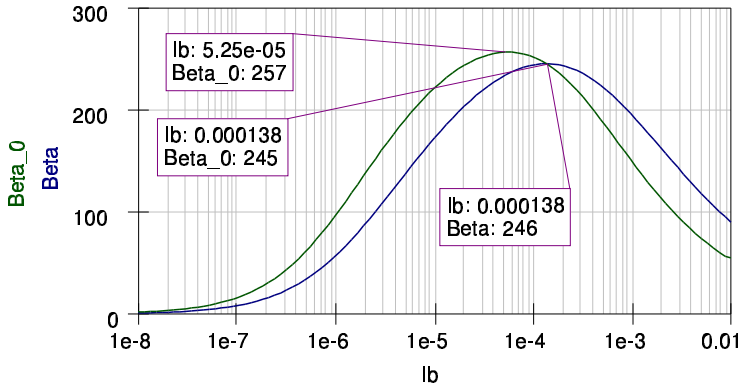


Figure 2.35: DC current gain vs. base current

It can be seen that the maximum AC current gain (257 @ 53µA) differs from the maximum DC gain. Also the AC current gain almostly equals the DC current gain at the base current for the maximum DC current gain. For maximum RF performance the base current with the maximum AC current gain could be chosen. But there may be other consideration, e.g. DC power dissipation, so we choose the bias point with the maximum DC current gain – arbitrarily.

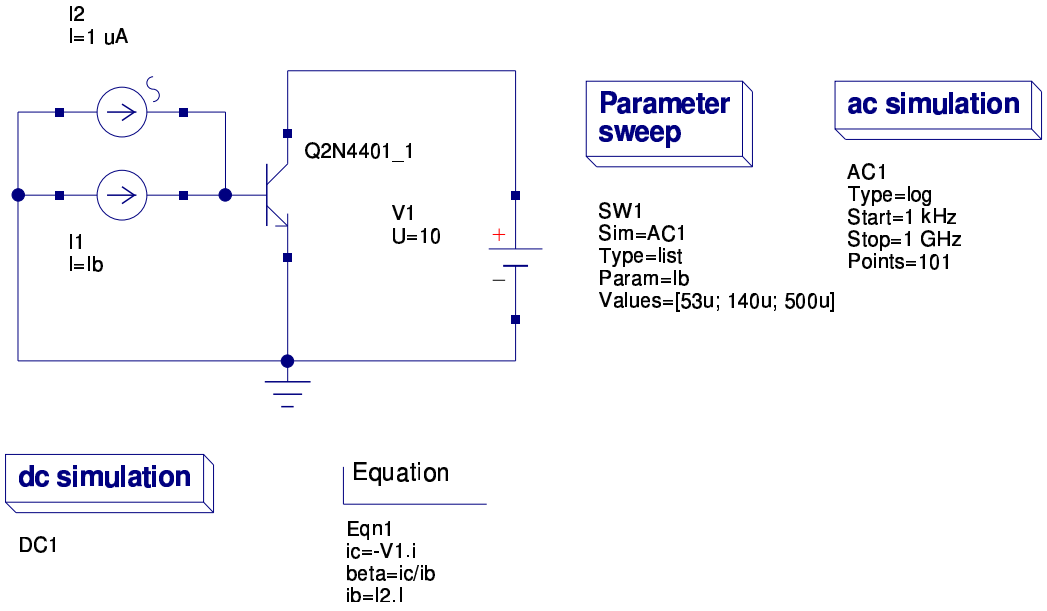


Figure 2.36: Bias dependent AC simulation setup

In fig. 2.36 is a DC bias dependent AC simulation setup shown. The DC base current **Ib** is swept for 53µA, 140µA and 500µA. Additionally the AC simulation block has been placed on the schematic.

The **Sim** parameter of the **SW1** parameter sweep is set to the instance name of the AC

simulation **AC1**. Qucs automatically "knows" that the DC simulation has to be run before each AC simulation since it is required to determine the appropriate bias points.

The AC current current source **I2** is in parallel to the DC current source and has an AC amplitude of 1µA. During the AC simulation the DC current source **I1** is an ideal open and the DC voltage source **V1** is an ideal short.

In the equations **V1.i** (mark the small i letter) denotes the AC current through the DC voltage source **V1**. The AC base current **ib** is taken from the input parameter **I2.I** denoting the value of the property **I** of the AC current source **I2** (1µA).

After pressing F2 – to start the simulation – the following cartesian diagram can be placed on the data display page, see fig. 2.37.

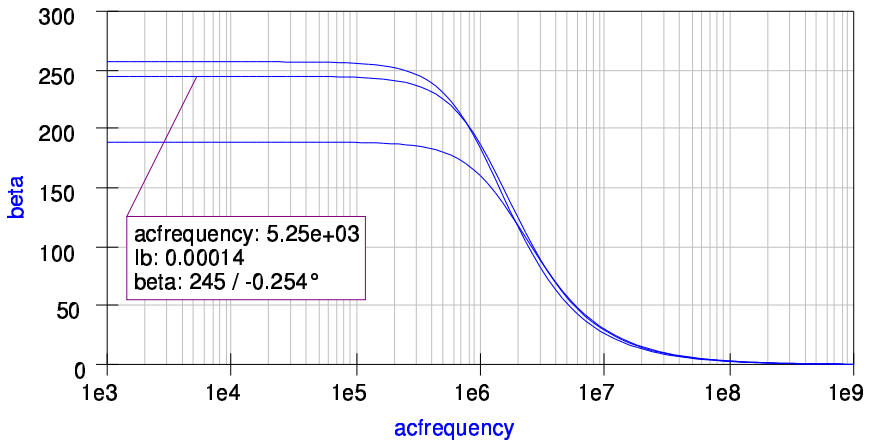


Figure 2.37: AC current gain of the bipolar transistor

The marker clearly shows for the low frequency range ($f \rightarrow 0$) the DC current gain of 246 (for $I_B = 140\mu A$) which was already determined in fig. 2.35.

In the next AC simulation setup shown in fig. 2.38 the parameter sweep is dropped to concentrate on the determination of the transit frequency. The transit frequency of a bipolar transistor denotes the frequency where the AC current gain drops to 1 (0 dB).

$$f_T \leftarrow |h_{21}|^2 = 1$$

Expressed in h-parameters of a general two-port the AC current gain is:

$$\beta_{RF} = h_{21} = \left.\frac{i_2}{i_1}\right|_{v_2=0}$$

whereas port 1 is the base and port 2 the collector. The side condition ($v_2 = 0$) is given in our setup since the DC voltage source is an ideal AC short.

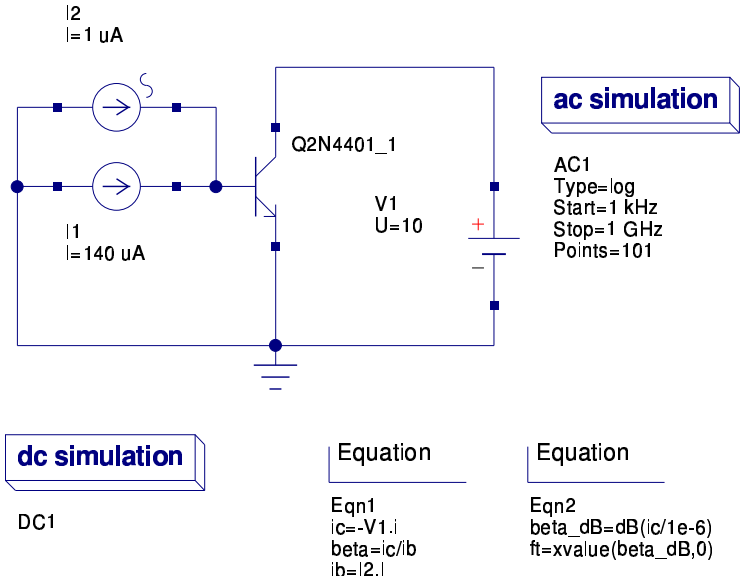
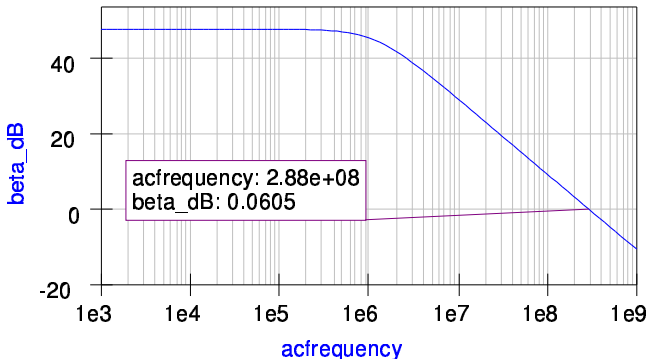


Figure 2.38: AC setup for determining the transit frequency

There are two more equations in the setup. One calculates the AC current gain in dB (which is $20 \cdot \log(\text{beta})$ and the other one is **ft=xvalue(beta_dB,0)**. The equation searches for the nearest given x-value (in this case the frequency) where **beta_dB** approaches **0**.



number	ft
1	2.884e+08

Figure 2.39: Bode plot of the current transfer function

In fig. 2.39 the Bode plot (double logarithmic plot) of the current transfer function of the bipolar transistor is shown. The current gain is constant up to the corner frequency and then drops by 20dB/decade. The marker finally denotes where the gain is finally 0dB. The equation for **ft** worked correctly as seen in the beside tabular. The transit frequency of the bipolar transistor in this bias point is approximately 288MHz.

2.3.4 AC simulation - A simple RC highpass

Simple circuit AC analysis (circuit frequency response analysis) can be carried out easily by using the *AC Simulation* block.

For instance, a simple high pass RC filter can be analyzed by constructing first the schematic displayed on figure 2.40 which corresponds to a high pass RC network.

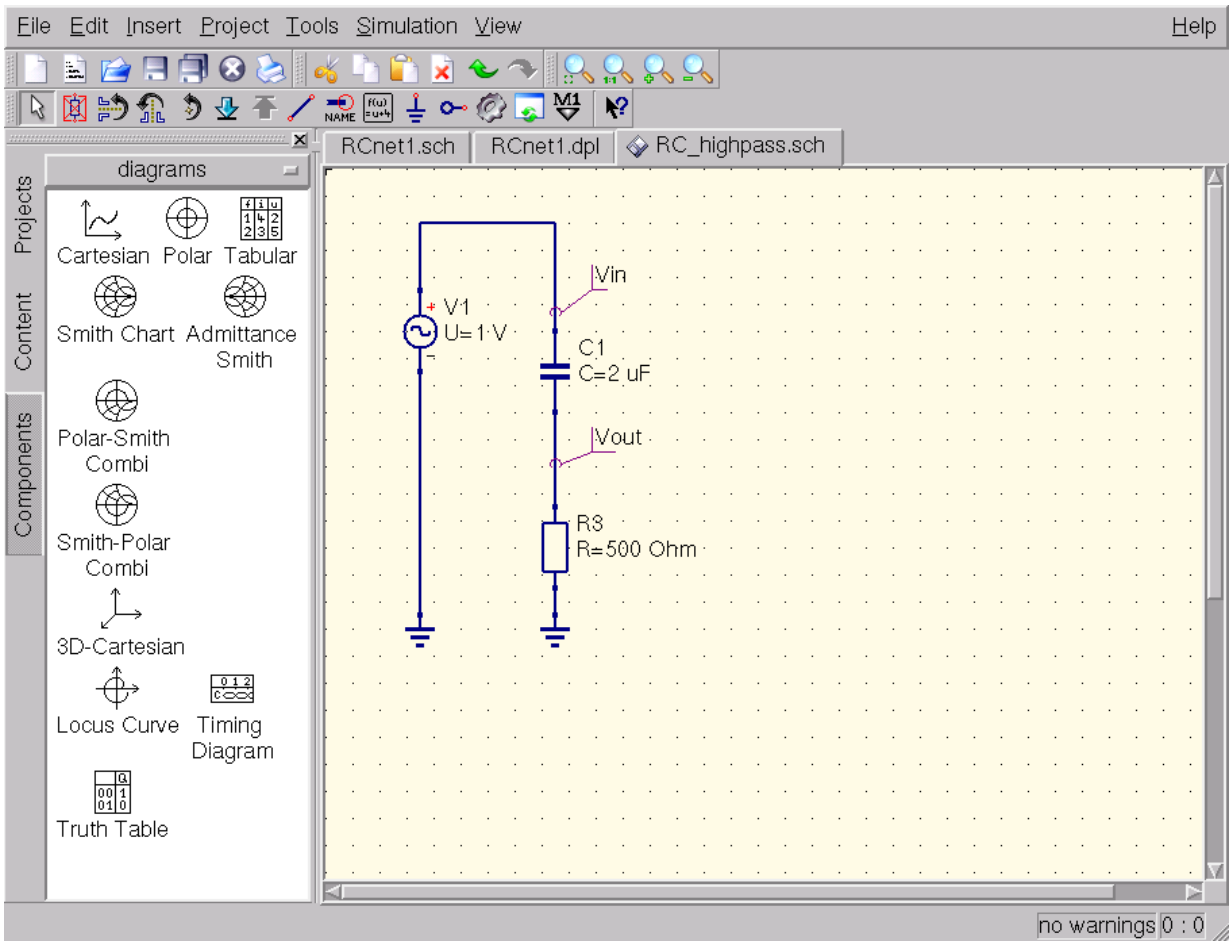


Figure 2.40: simple RC high-pass filter schematic

Performing the actual AC analysis is as easy as dragging and dropping an *AC Simulation* block available under the *Simulations* tab as can be seen in figure 2.41.

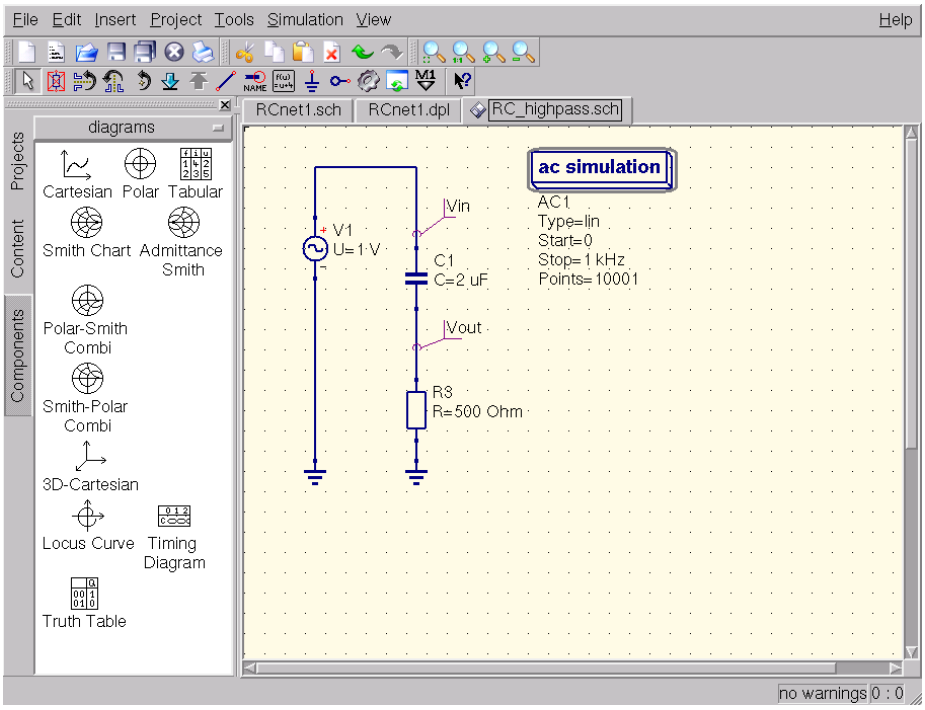


Figure 2.41: AC simulation block placed

Once this is done one must configure the ranges of the simulation analysis by clicking twice on the *AC Simulation* box as can be seen in figure 2.42.

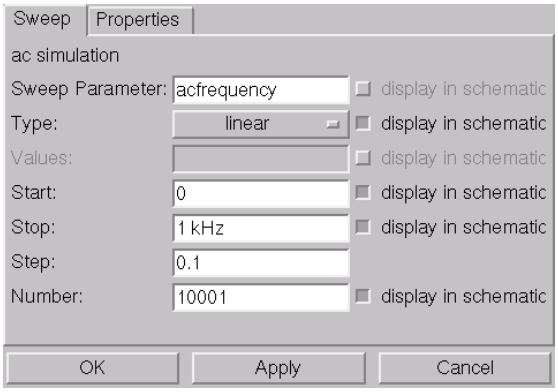


Figure 2.42: AC simulation block configuration dialog

Finally by pressing F2 the simulation takes places and a graphic report can be generated by selecting the right plot as seen in the previous sections. The final view of the network with its respective frequency analysis can be seen on figure 2.43.

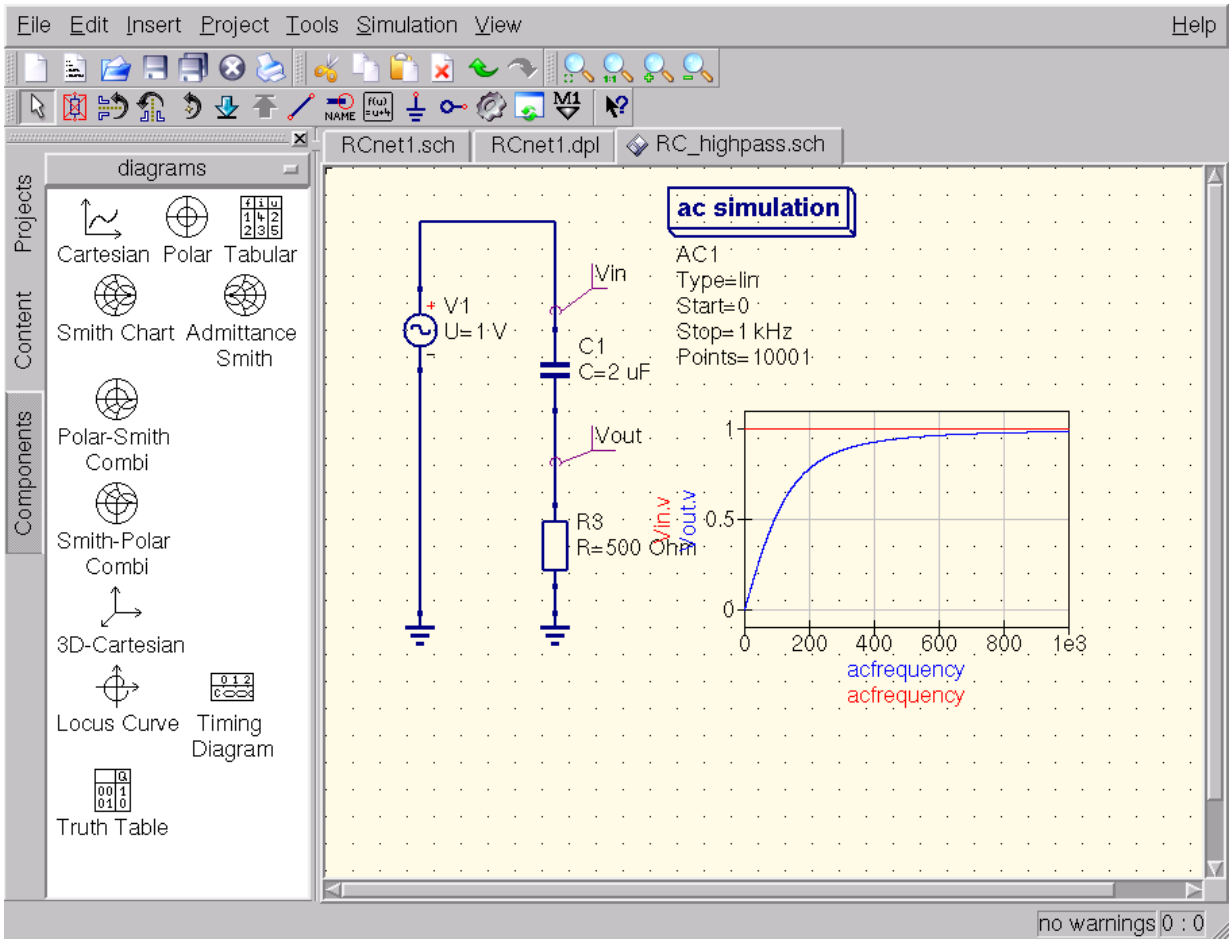


Figure 2.43: AC simulation results

2.3.5 Transient simulation - Amplification of a bipolar transistor

Based on the schematic in fig. 2.38 we are now going to simulate the bipolar transistor in the time domain.

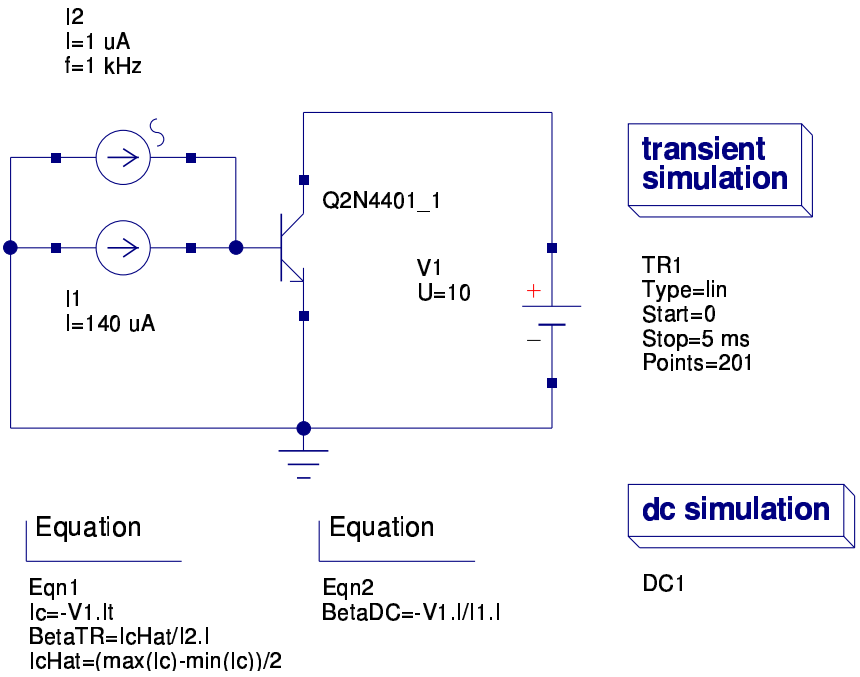


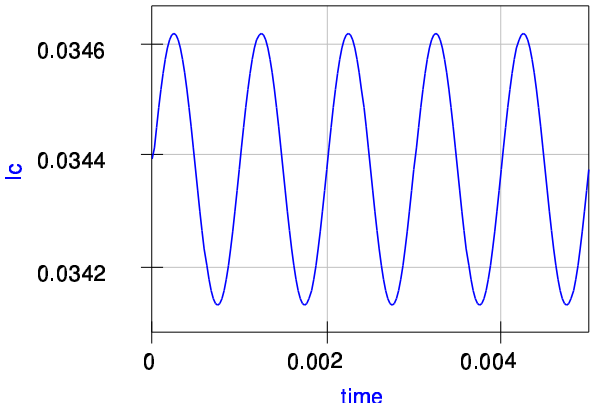
Figure 2.44: Transient simulation setup

As shown in fig. 2.44 the transient simulation block was placed on the schematic. Also the frequency **f** of the AC current source **I2** was set to 1kHz. The start time of the transient simulation is set to 0 and the stop time to 5ms which will include 5 periods of the input signal.

The additional DC simulation block is not necessary for the transient simulation but left there for some result comparison.

The collector current in the equations is denoted by the transient current **-V1.It**. The peak value if the collector current is determined by the equation for **IcHat**. The current gain during transient simulation is calculated using **BetaTR=IcHat/I2.I** whereas **I2.I** denotes the component property **I** of the the current source **I2** (which is 1µA peak). The current gain **BetaDC** is computed for convenience.

The equation blocks imply that the order of appearance of assignments does not matter (e.g. **IcHat** is used before computed). The equation solver will take care of such dependencies.



number	IcHat	BetaTR	BetaDC
1	0.000245	245	246

Figure 2.45: Transient results

Fig. 2.45 shows the results of the transient as well as DC simulation. The time dependent collector current oscillates around its bias point. The current gain of the transient signal corresponds perfectly with the DC value. That is because a rather small frequency of 1kHz was chosen.

2.3.6 S-parameter simulation - Transit frequency of a BJT

In the following section the S-parameter simulation is introduced. The S-parameter simulation is – similar to the AC simulation – a small signal analysis in the frequency domain.

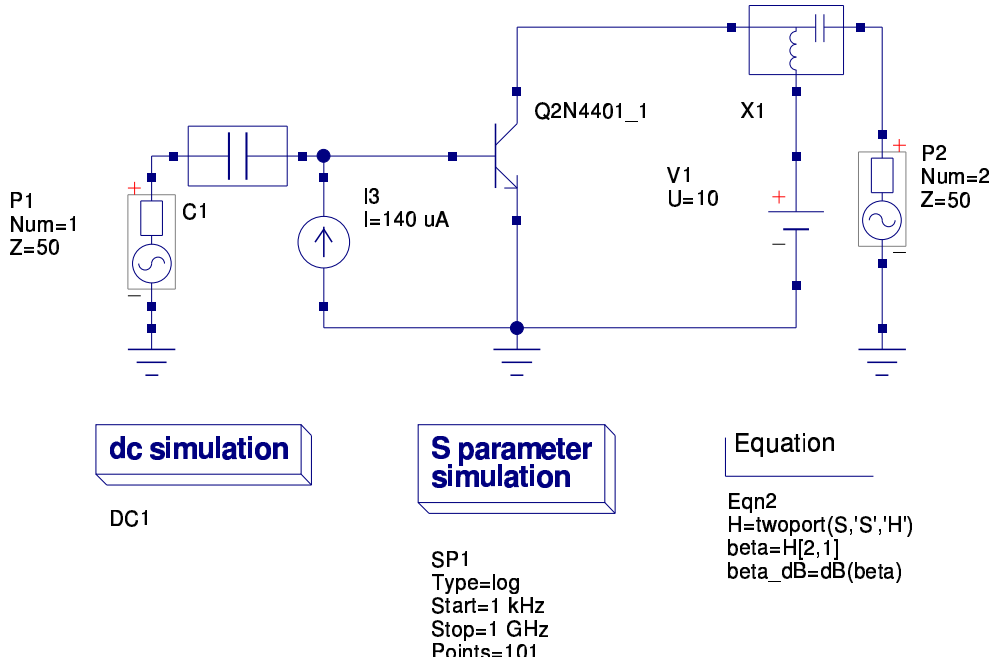


Figure 2.46: S-parameter simulation setup for the bipolar transistor

Similar to the AC setup in fig. 2.38 the S-parameter setup in fig. 2.46 uses the same biasing. The setup will be used to determine the transit frequency of the bipolar transistor.

The two AC power sources **P1** and **P2** are required for a two-port S-parameter simulation. They can be found in the **Components** tab in the **sources** category. Depending on the number of these kind of sources one-port, two-port and multi-port simulations are performed. The **Num** property of the sources determines the location of the matrix entries in the resulting S-parameter matrix. The **Z** properties define the reference impedance of the S-parameters.

The additional DC block **C1** at the base node and the bias tee **X1** on the collector is used to decouple the signal path of the biasing DC sources from the internal impedance of the AC power sources. Also the bias tee ensures that the AC signal from the **P2** source is not shorted by the DC source **V1**. The same functionality is achieved by the DC current source **I3** at the base. It represents an ideal AC open.

The S-parameter simulation itself is selected by placing the S-parameter block **SP1** on the schematic. The same frequency range is chosen as in the previous AC simulations.

The equations contain a two-port conversion function which convert the resulting S-parameter **S** into the appropriate H-parameters **H**. Again the AC current gain h_{21} is calculated and converted in dB.

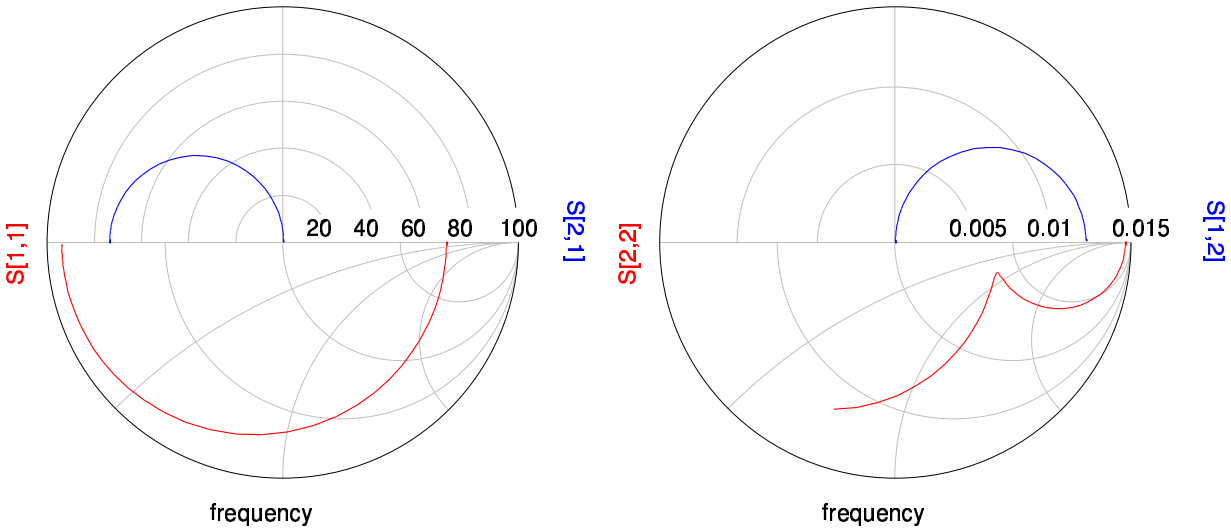


Figure 2.47: S-parameters of the bipolar transistor

In fig. 2.47 the four complex S-parameters are displayed in two **Polar-Smith Combi** diagrams. They represent what can be expected from a typical bipolar transistor.

Using the computed H-parameters we can now compare the S-parameter simulation results with those of the AC simulation. Fig. 2.48 shows that the curves **beta_dB** of both simulation setups cover perfectly each other. Again the transit frequency is approximately 288MHz.

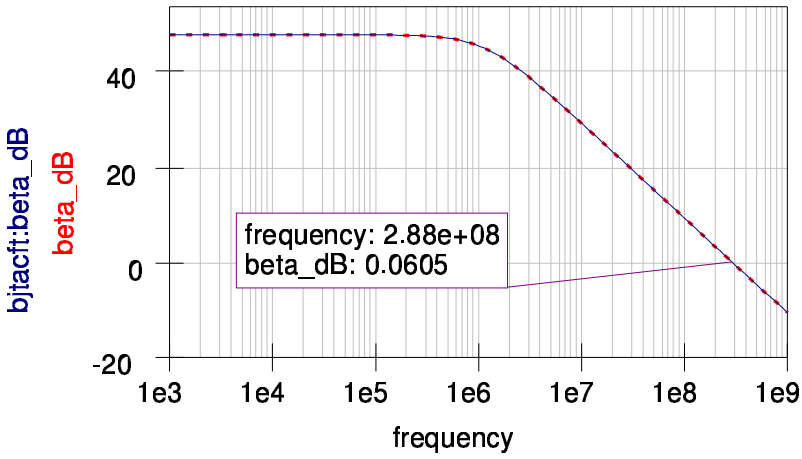


Figure 2.48: Comparison between S-parameter and AC result

The diagram implies that you can compare data curves from different setups. This is indicated by the **bjtacft:** prefix. The appropriate dataset file **bjtacft.dat** can be selected in the diagram dialog as shown in fig. 2.49.

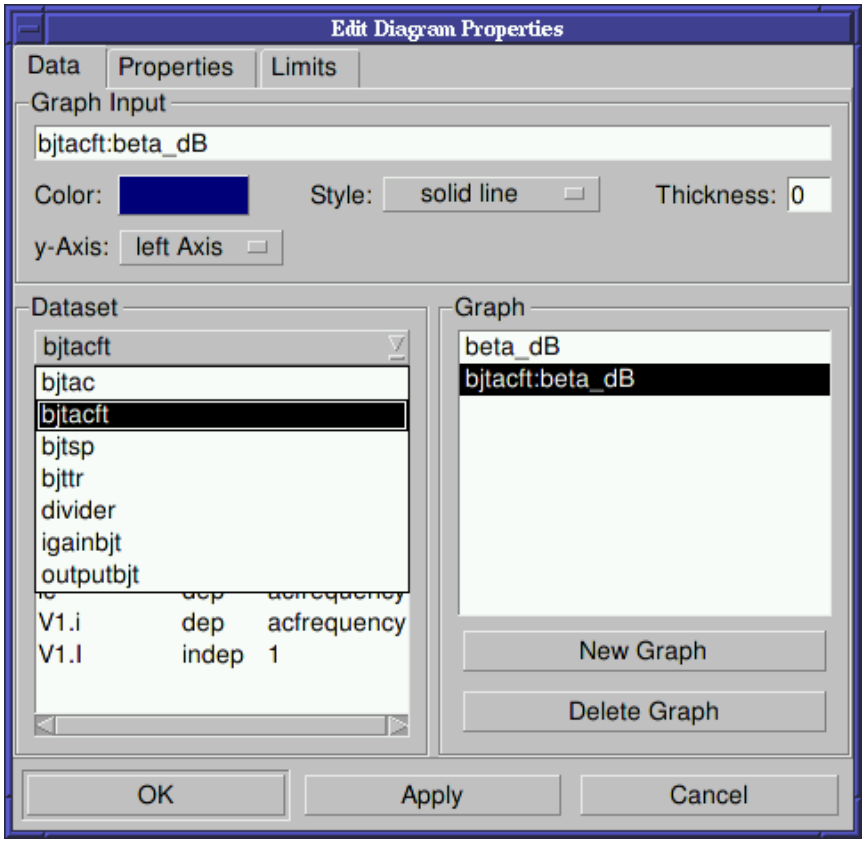


Figure 2.49: Choosing graphs from different datasets

The current S-parameter setup is called **bjtsp** and the setup shown in fig. 2.38 was called **bjtacft**. Please note that only datasets from the same project can be compared with each other.

2.3.7 S-parameter and AC simulation - A Bessel band-pass filter

The interested reader may have noticed that there seems to be a relationship between AC analysis and the S-parameter simulation. In the next section we are going to explain this relationship using a simple filter design.

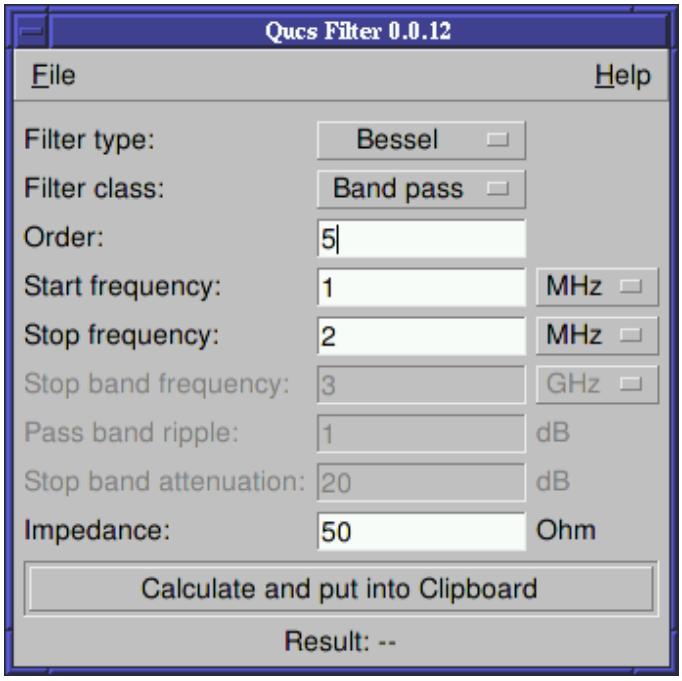


Figure 2.50: Filter synthesis application

In fig. 2.50 the filter synthesis program coming with Qucs is shown. You can start it by the Ctrl + 2 shortcut or by choosing the **Tools** → **Filter synthesis** menu entry. The user can choose between different types of filters and the filter class (lowpass, highpass, bandpass or bandstop). Also the appropriate corner frequencies and the order must be configured. When setup correctly you press the **Calculate and put into Clipboard** button. The program will indicate if it was possible to create the appropriate filter schematic. If so, the application passes the schematic to the system wide clipboard.

Back in the schematic editor you can paste the filter design into the schematic using the Ctrl + V shortcut or by choosing the **Edit** → **Paste** menu entry.

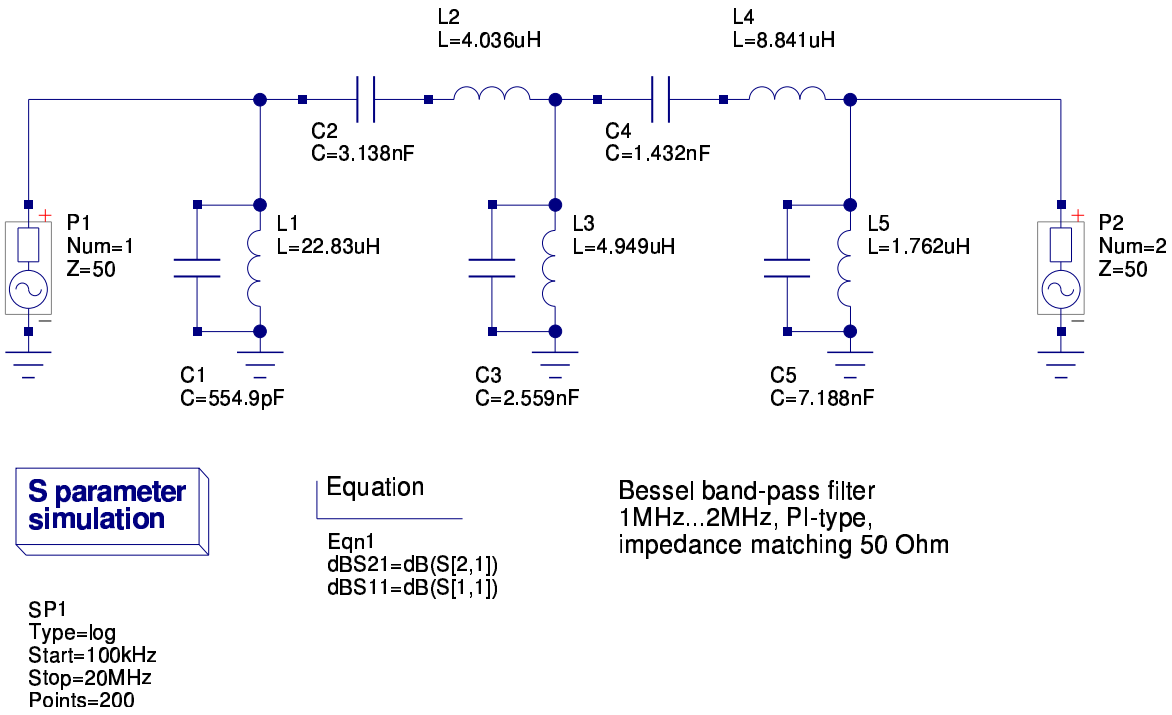


Figure 2.51: Schematic for 5th order Bessel band-pass filter

The schematic shown in fig. 2.51 was automatically created by the filter synthesis program and can be simulated as is. It contains the LC-ladder network forming the actual filter, the two S-parameter ports (the AC power sources) as well the S-parameter simulation block with the appropriate frequencies pre-configured. Additionally there is an equation computing the transmission and reflection of the filter network in dB.

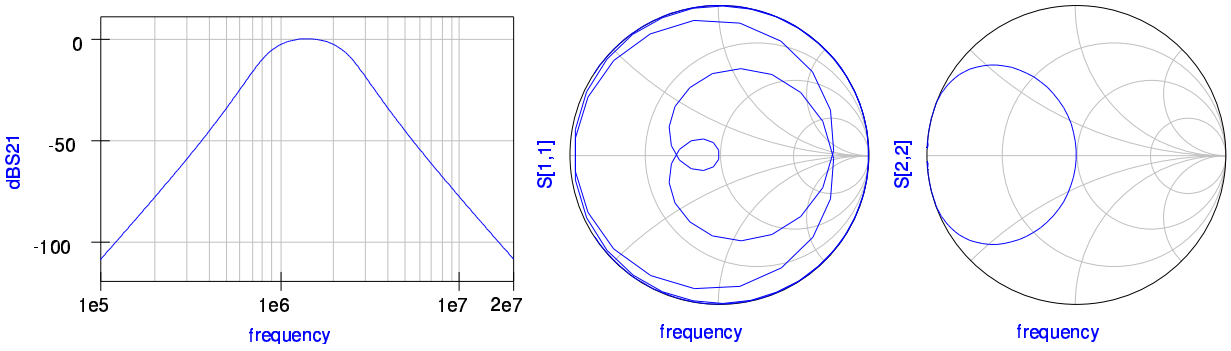


Figure 2.52: S-parameters of the band-pass filter

The results of the S-parameter simulation are depicted in fig. 2.52. In the logarithmic cartesian diagram the transmission of the filter clearly shows the band-pass behaviour between the selected frequencies 1MHz and 2MHz. Additionally the input- and output reflections can be seen in the two Smith charts.

Now two AC setups will be created to calculate the same S-parameters as found in the previous simulation. In fig. 2.53 the LC-ladder network is unchanged but the S-parameter ports are replaced by a 50Ω resistor and an AC voltage source in series. Also there is now an AC simulation block with the same frequency sweep chosen as in the previous S-parameter simulation.

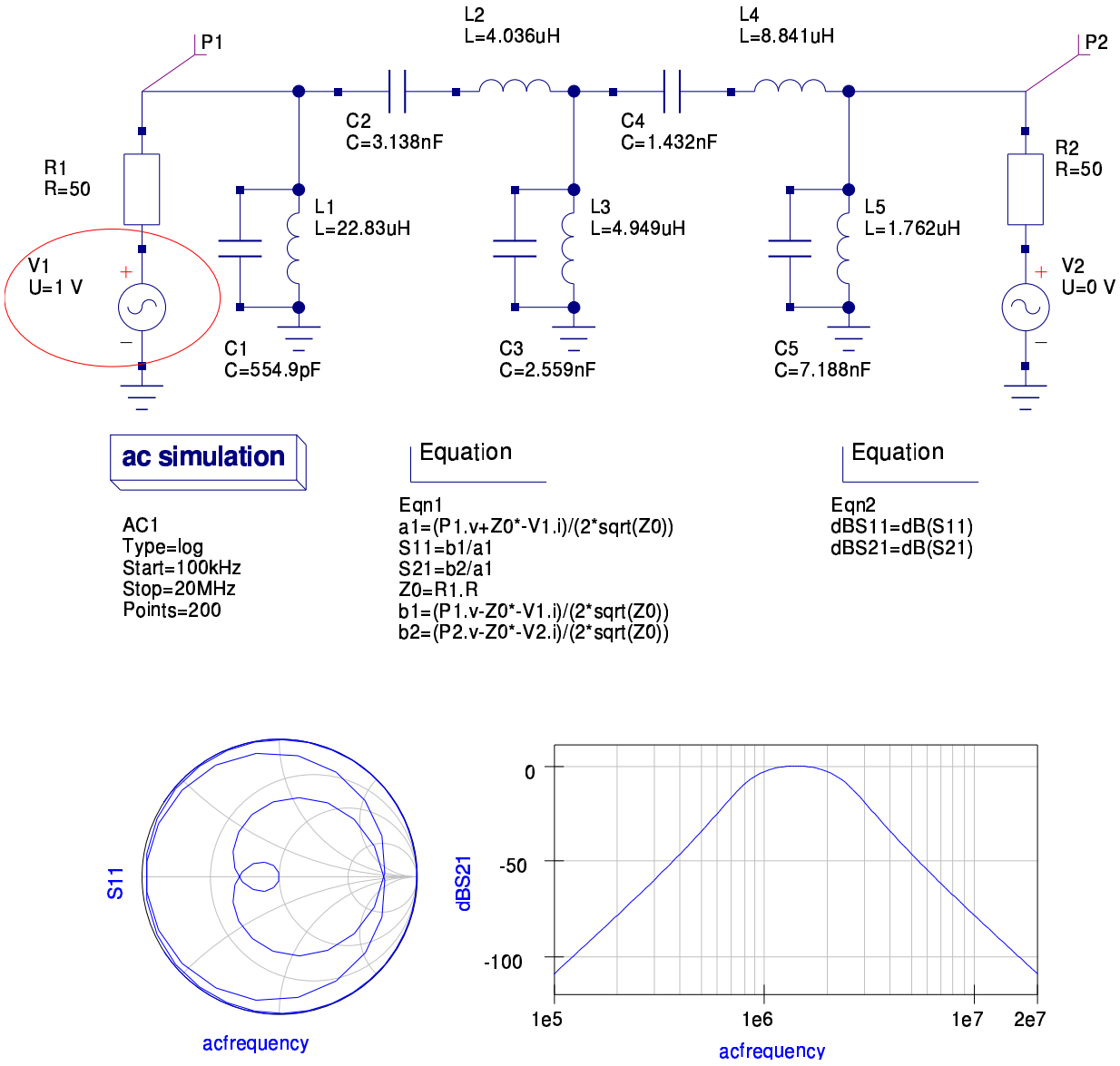


Figure 2.53: S-parameters at port 1 of the band-pass filter using AC analysis

At this point some theory must be stressed.

S-parameters are defined by ingoing (a) and outgoing (b) power waves:

$$a = \frac{V + Z_0 \cdot I}{2 \cdot \sqrt{Z_0}}$$
$$b = \frac{V - Z_0 \cdot I}{2 \cdot \sqrt{Z_0}}$$

whereas Z_0 denotes the reference impedance the S-parameters will be normalized to. With this definition the two-port S-parameters can be written as:

$$S_{11} = \left.\frac{b_1}{a_1}\right|_{b_2=0} \qquad S_{21} = \left.\frac{b_2}{a_1}\right|_{b_2=0} \qquad S_{22} = \left.\frac{b_2}{a_2}\right|_{b_1=0} \qquad S_{12} = \left.\frac{b_1}{a_2}\right|_{b_1=0}$$

Back at the schematic in fig. 2.53. The amplitude of the AC voltage source **V1** is set to 1V (but can be any other value different from zero) and the side condition $b_2 = 0$ is fulfilled by setting the amplitude of the AC voltage source **V2** to 0V. The additional equations just calculate the S-parameters as they are defined from the AC simulation values.

Please note the current directions through the AC voltages sources **V1.i** and **V2.i**. They must be considered by the unary minus in the equations.

The results of this simulation again show the filter transmission function as we already know it from the S-parameter simulation. Also the reflections at port 1 look identical.

In the second schematic shown in fig. 2.54 the second port is handled. The amplitude of the AC voltage source **V2** is set to 1V and the side condition $b_1 = 0$ considered by a zero AC voltage source **V1**. Again the appropriate equations are used to compute the two remaining S-parameters.

The below simulation results again verified that we can perform a partial S-parameter analysis using the AC simulation block and some additional equations. The diagrams in fig. 2.54 and fig. 2.52 are identical.

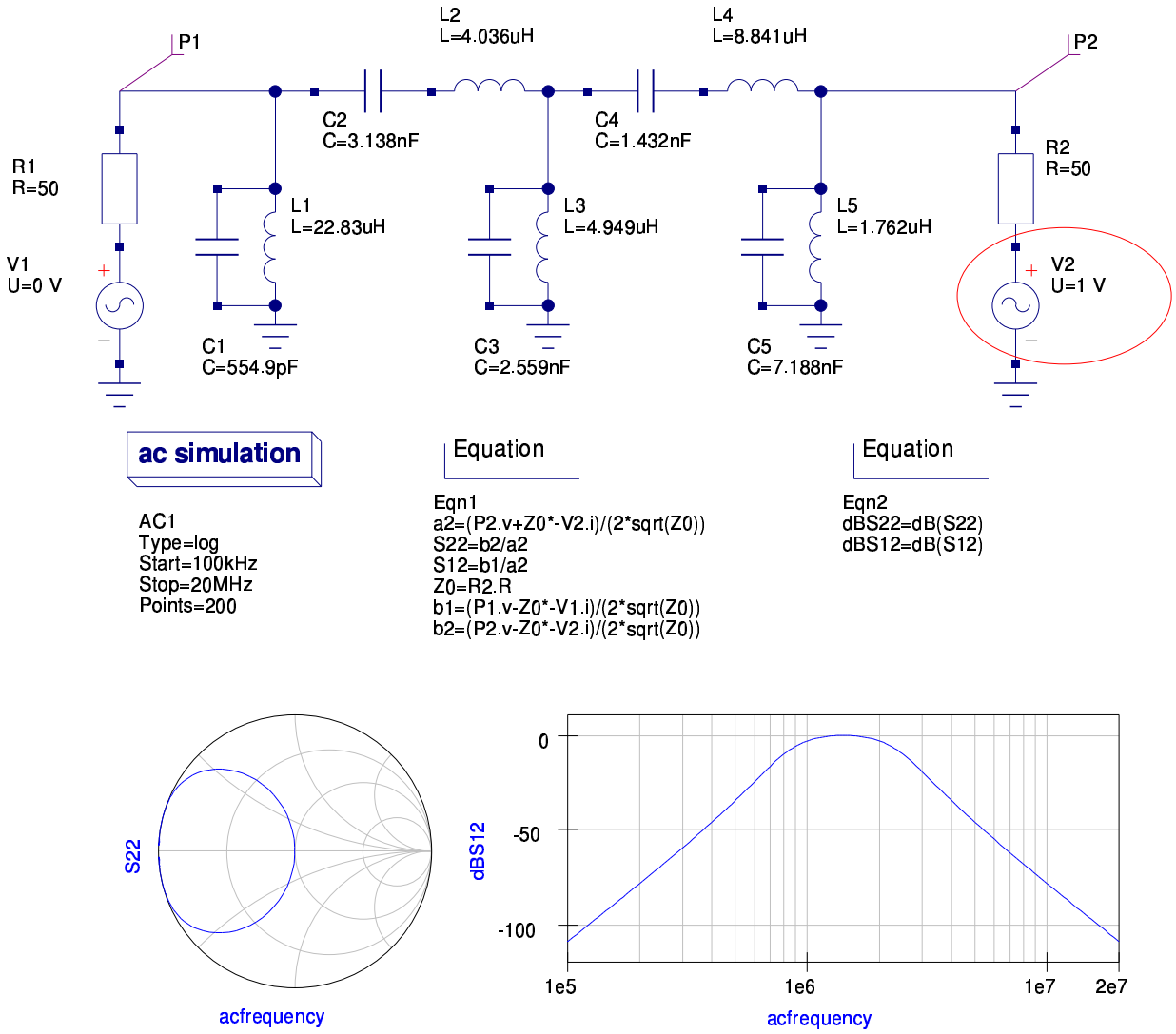


Figure 2.54: S-parameters at port 2 of the band-pass filter using AC analysis

Recapitulating we learned from this example that a S-parameter simulation is a number of AC simulations with some additional calculation formulas. This is true though the actual simulation algorithms implemented in Qucs are completely different.

3 Understanding RF Data Sheet Parameters

. . . prepared by Norman E.Dye from Motorola RF Division : AN 1107[1]. Since this AN is essential to our topics, it is good to make a small reference to it. All AN from Motorola are a reference is this field. This chapter is only an extract, but the main points are hilighted herein. . . .
The author.

3.1 Introduction

Data sheets are often the sole source of information about the capability and characteristics of a product. This is particularly true of unique RF semiconductor devices that are used by equipment designers all over the world. Because the circuit designer often cannot talk directly with the factory, he relies on the data sheet for his device information. And for RF devices, many of the specifications are unique in themselves. Thus it is important that the user and the manufacturer of RF products speak a common language, what the semiconductor manufacturer says about his RF device is understood fully by the circuit designer.

This paper reviews RF transistor and amplifier module parameters from maximum ratings to functional characteristics. It is divided into five basic sections:

1. DC specifications,
2. power transistors,
3. low power transistor,
4. power modules,
5. linear modules.

Comments are made about critical specifications about how values are determined and what are their significance.

[1]This note could be found on old application notes databook from Motorola, if you have one keep them, it is a real treasure.

3.2 DC specifications

Basically, RF transistors are characterized by two types of parameters: DC and functional. The "DC" specs consist of breakdown voltage, leakage current, h_{FE} (DC β) and capacitances, while the functional specs cover gain, ruggedness, noise figure, Z_{in} and Z_{out}, S parameters, distortion, etc .... Thermal characteristics do not fall cleanly into either category since thermal resistance and power dissipation can be either DC or AC. Thus we will treat the spec of thermal resistance as a special specification and give it its own heading called "thermal characteristics".

3.3 Maximum ratings and thermal characteristics

4 DC Analysis, Parameter Sweep and Device Models

4.1 DC Static Circuits

A favourite question in electronics courses used to be:

> *You have twelve one ohm resistors; you connect them together so that each resistor lies along the edge of a cube. What is the resistance between opposite corners of the cube?*

The intention may have been to teach soldering, as more than one student solved it by making just such a cube! These days we can do that without touching the soldering iron; we simulate the circuit.

Here is my attempt to make a cube in Qucs; anyone is welcome to try and improve it.

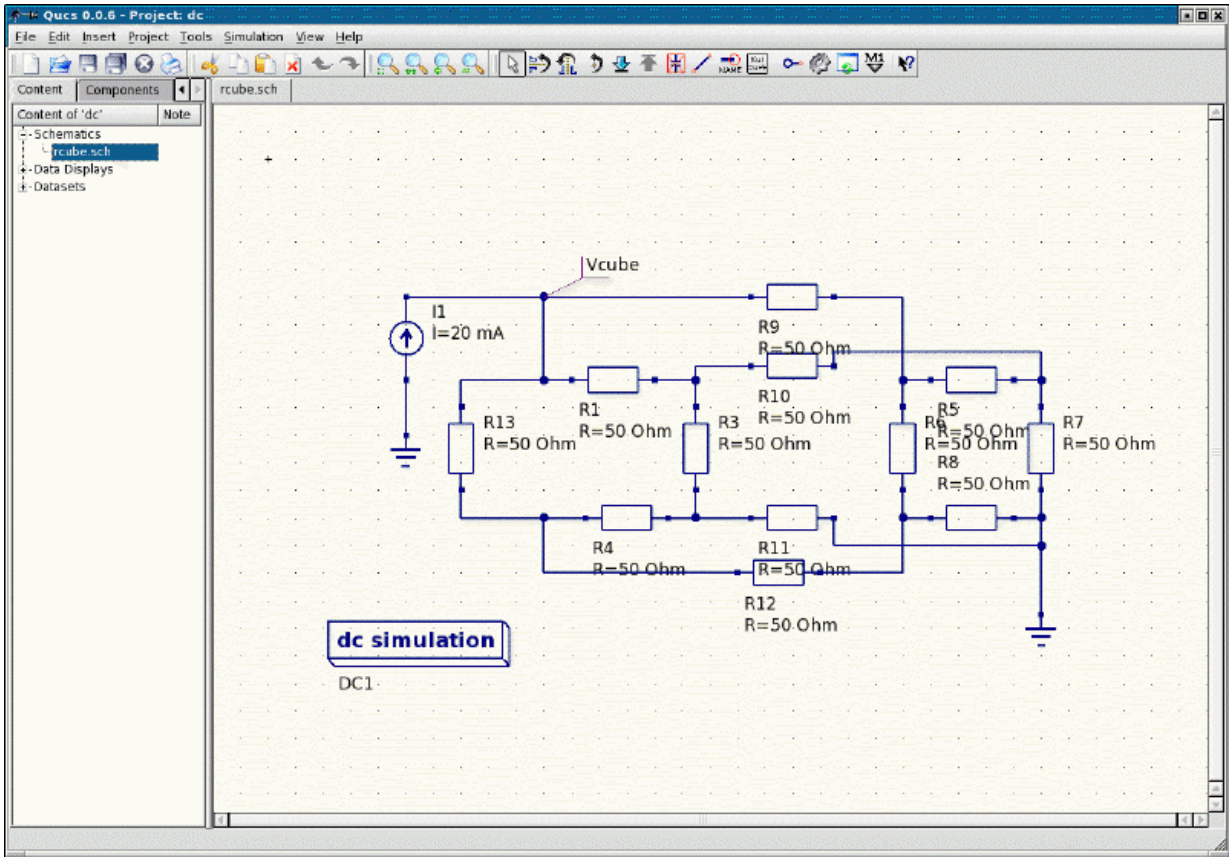


Figure 4.1: resistor cube schematic

All I did was select resistance in the left hand component window and paste them down, rotating as necessary, until I had twelve on the schematic. Then I wired two sets of four into squares, then connected the remaining four between the corners of the squares. Which I'm sure is topologically the same as a cube.

Which all might seem trivial, but is a good reminder right at the beginning that we are creating a virtual representation of a physical circuit. Sometimes we have to bend and squeeze things to get it into a format that our simulator will accept, which leaves us wondering whether we are working with an accurate representation.

The Rule is: if we can correlate the junctions of our components with those of the real circuit, we are accurately representing the physical circuit. And, I might add, it is ALWAYS worth checking that we have done it right; simulate the wrong circuit and it will tell you lies.

With my cube of resistors accurately drawn, I only have to hit the simulation button and the tabulated results will show me the voltage at the corner node. As I am forcing a constant current through the cube from one corner to another, Ohm's Law tells me that

the voltage between those corners will give me the resistance. If I use a current of one amp, the output voltage will be equal to the resistance in ohms.[1]

Those with good attention to detail will be complaining about now that I haven't really solved the problem, as the question mentioned one ohm resistors while I have used fifty ohms. Well, yes, I cheated. Which I often do in simulations.

To set all the resistances to the correct value I would have had to open the Properties Editor window twelve times; here is how it looks...

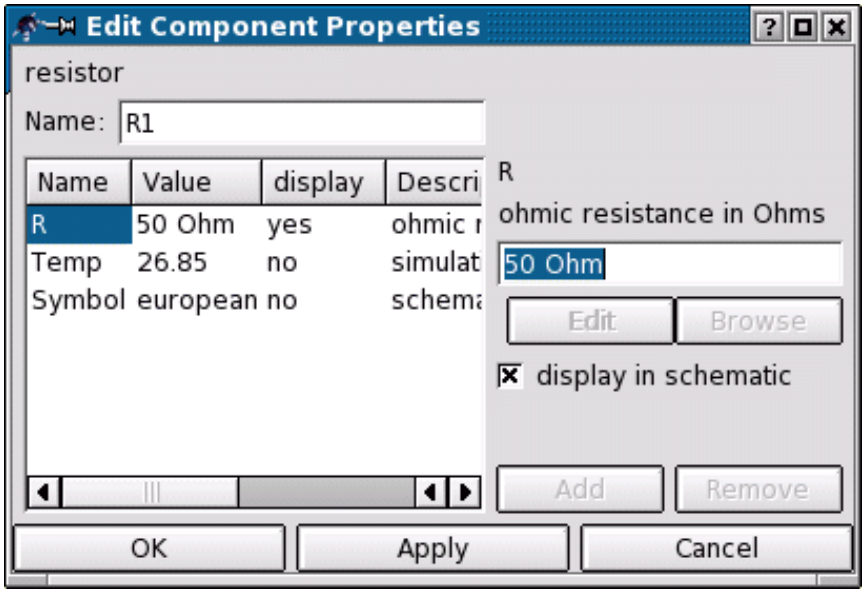


Figure 4.2: component property dialog

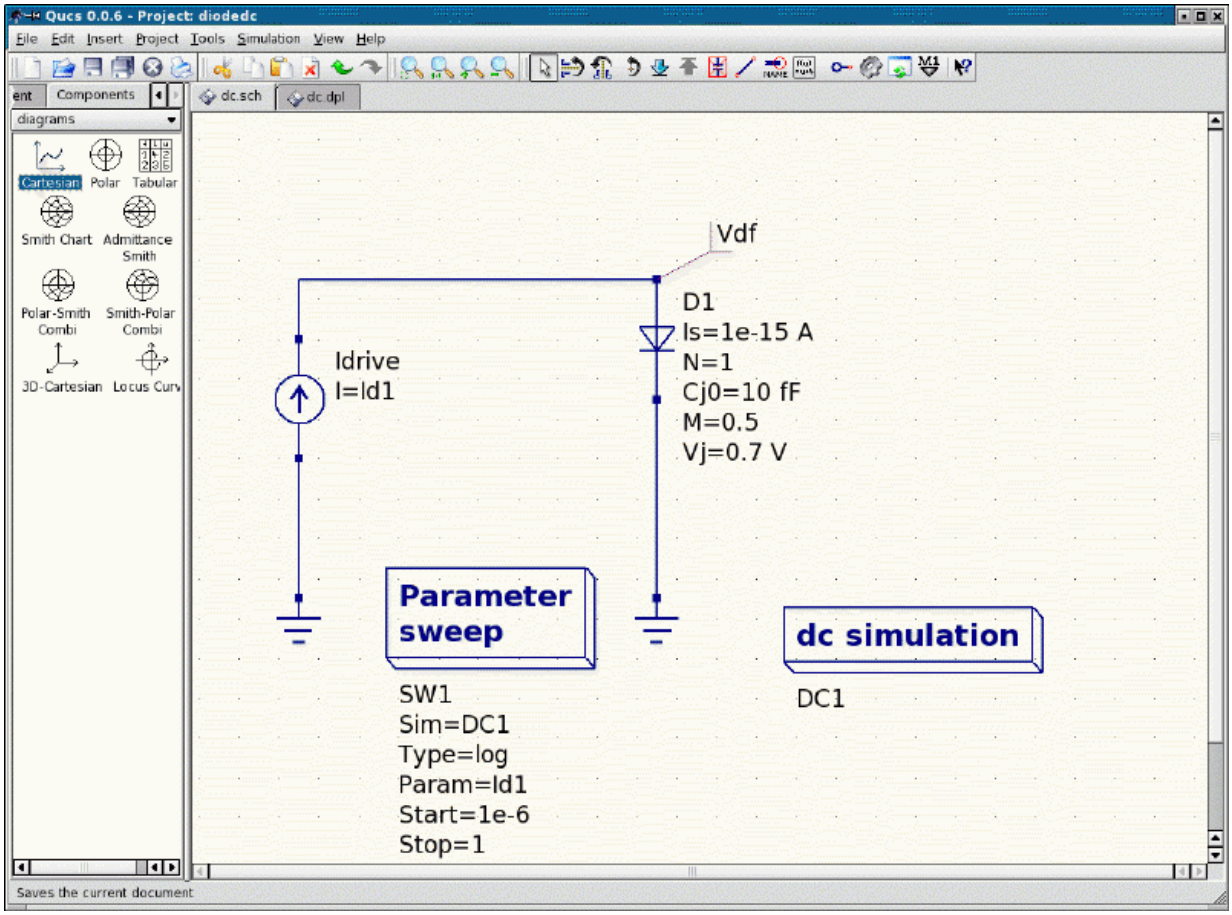
and the highlighted value is inviting me to type in an alternative. I could have done this, but natural laziness got the better of me. I reasoned that fifty ohms is fifty times too high, but if I reduced the current source from one amp to twenty milliamps, the output voltage would be the same. You will find such laziness (or acute perception, depending on is telling the story!) can save much time and effort.

4.2 When Things Vary

All of which is interesting, but not nearly as interesting as when we start changing things like the supply voltage and see the effects. For linear devices with a DC supply, the answer would be: not much. It's when we introduce non-linear elements that things start to happen.

[1] I could tell you the value my simulation gave, but why should I spoil your fun.... go ahead and run it yourself. If you really want to be thorough you could then also build the circuit and measure the result.....

The simplest non-linear element is the diode, and the question we ask most often about a diode is: how does the diode forward voltage vary with current? So back to Qucs and draw this circuit...

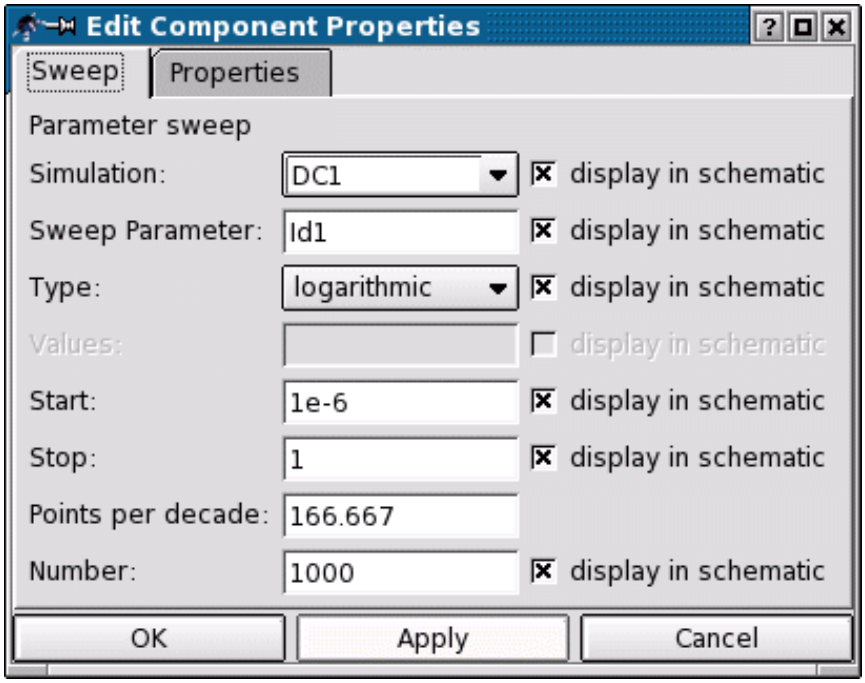


This circuit looks deceptively simple, but it introduces a few more features of Qucs, so let's go through them in order.

The components were again selected from the left hand window and wired together. Then the two boxes were selected from the **simulations** window.

The DC simulation box can be pretty much left as is for now, but take note of the name of the simulation: **DC1**.

The **Parameter sweep** box properties dialog looks like this when opened...

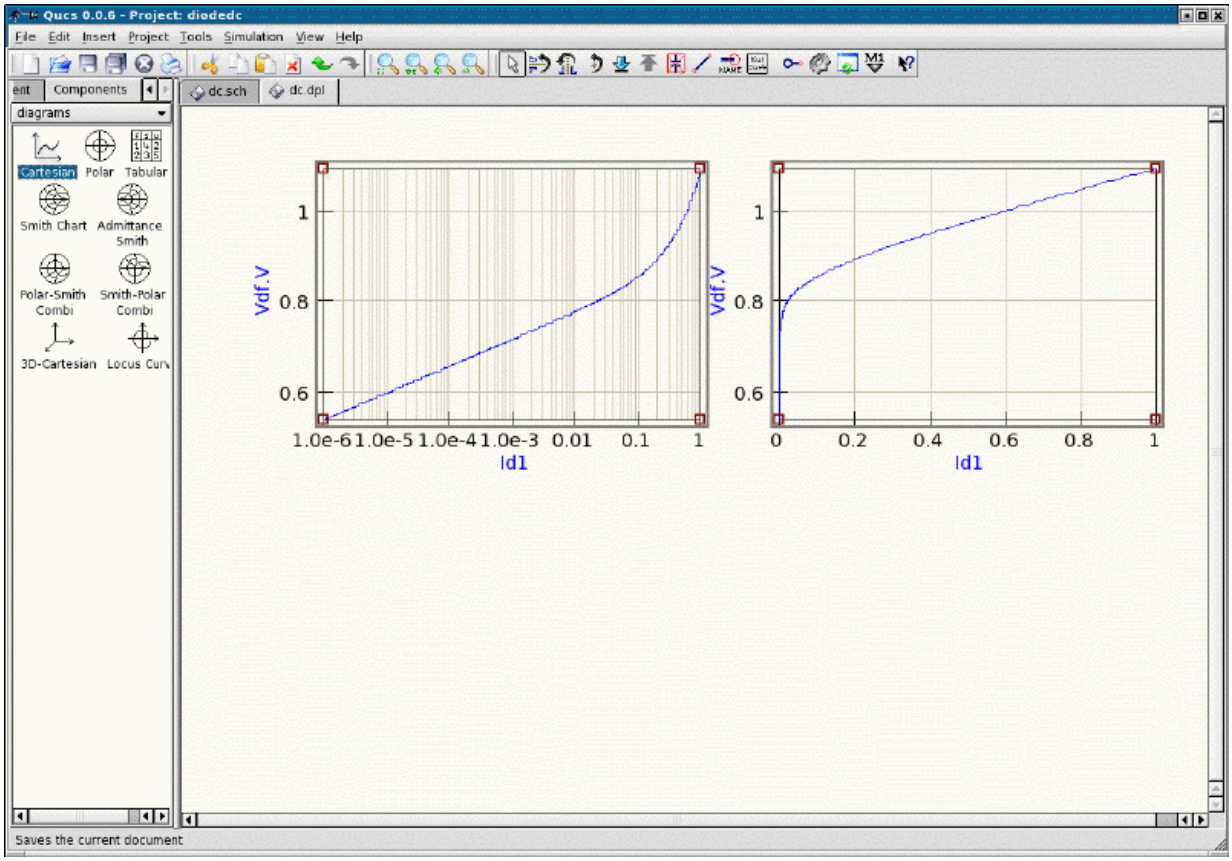


The first two items to take note of are the **Simulation** entry (here **DC1**, corresponding to the name of the simulation box) and the **Sweep Parameter** entry, here entered as **Id1**. If you look at the current source driving our diode you will see that it just happens to be labeled **Idrive**. So the result of all this is that the component property value **Id1** of the current source's property **I** will be swept through a range of values as determined by our parameter sweep function named **SW1**.[2]

The rest of the entries set the type of sweep (here logarithmic) and the range of values over which to sweep. You can try different values in any of these to see the effect; one of the advantages of a simulator over a physical prototype is that you can't blow up your diode by feeding too much current through it!

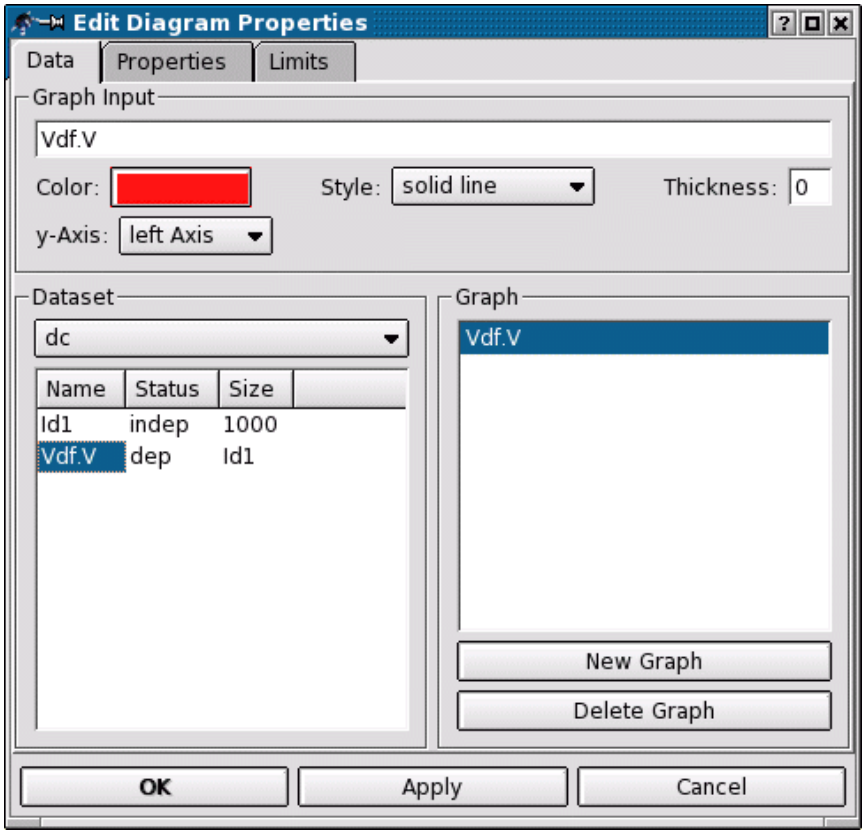
So I hit the simulation button and it passed me over the results page, and I created a couple of graphs of the output. This is how my screen looked...

[2]You can change this name if you wish, in the Properties menu of the Edit properties window.



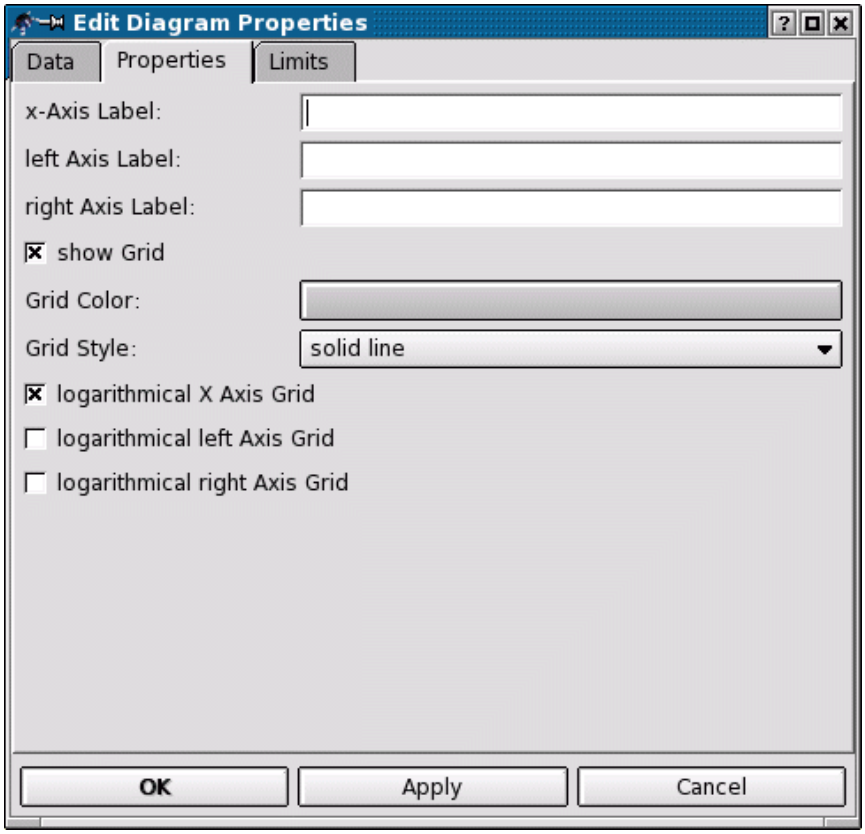
In each case I have a plot of diode forward voltage (Y-axis) against forward current (X-axis). The left hand graph has a logarithmic scale for forward current, while the right hand graph uses a linear current scale. How did I do that? Well, you should know by now that all things are easy with Qucs!

When you select a graph type from the left hand window and drag it into the viewing area, it creates a graph and opens a dialog which looks like this



The left hand window shows the available variables and whether they are dependent or independent. Here the current Id1 is the independent variable, and the forward voltage Vdf.V is the dependent. Double-click on the entry for Vdf.V and it is transferred to the right hand side; hit OK and the graph will be drawn.

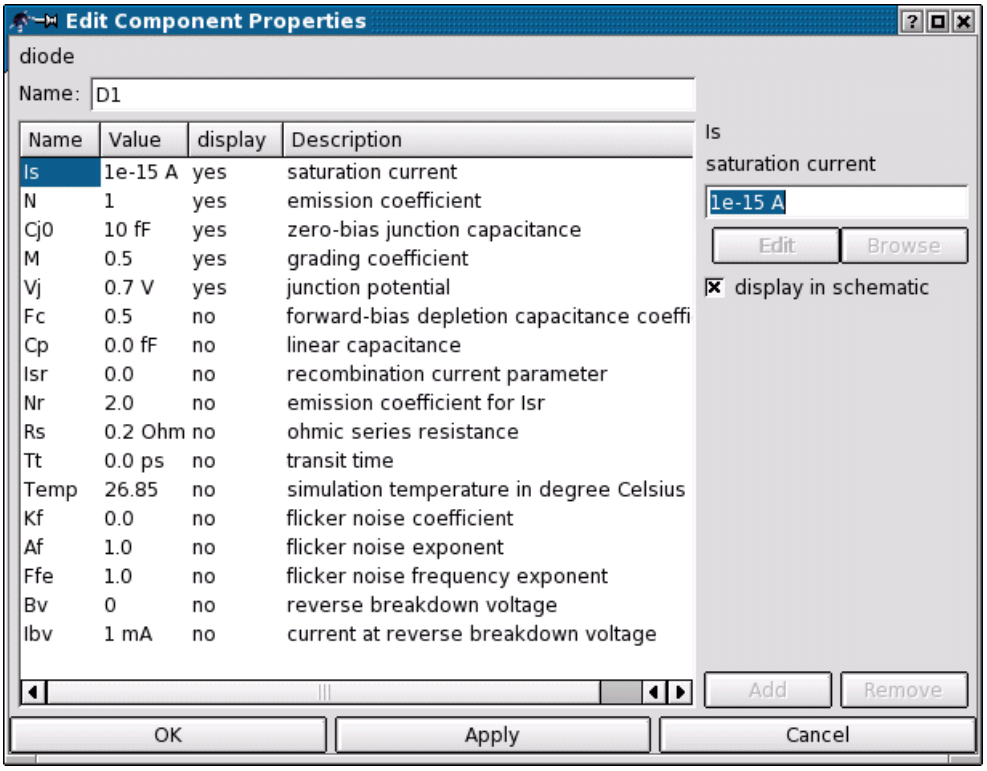
That should give you something like the right hand graph in my screenshot above. Do it all again, but this time before clicking OK open the Properties window, which looks like this.



Here I've selected a logarithmic X Axis, which gave me the graph on the left hand side. I've also moved them around and re-sized them to pretty them up; you can do all kinds of fancy things if you want.

Now I've sneaked in another test to see if you are really following this. Those of you who did run this simulation are probably wondering about now why your graphs look rather different to mine. In particular, at high currents on the logarithmic scale your curve is a straight line, while mine curves upwards alarmingly. What is happening ?

What I did was open the Properties dialog for the diode and set some parameters. This is what the dialog box looks like...



and each of these entries sets one parameter of the virtual component we are using to model the diode.

So, what are these parameters? Time to explore one of the delights of computer circuit simulation, device modeling...

4.3 Models and Parameters

When the computer creates that small piece of virtual reality which represents your physical circuit, it uses sets of equations which describe the operation of each device you insert. The equation which relates the diode DC forward voltage as a function of current is

$$I_d = I_s \cdot \left(e^{\frac{V_d}{n \cdot V_t}} - 1 \right)$$

where V_t is the forward voltage drop at 25 degrees C of an ideal junction, also given by

$$V_t = \frac{k_B \cdot T}{q}$$

where

$$k_B = \text{Boltzmann's constant}$$
$$T = \text{temperature in degrees Kelvin}$$
$$q = \text{charge of the electron}$$

most of these are constants that the program already knows about. The ones we need to supply are the ones listed in the properties editor window. For the DC characteristics, most of the time, the only ones we need to worry about are Is, the saturation current, and T, the temperature. If we are going to push relatively high currents through the diode we can also include an estimate for the series resistance Rs; if we are worried about low current behaviour then we need to add the reverse current parameter Isr.

How do we know what values to insert? Much could be written about device modeling; much indeed has been written about device modeling. As always, we really have two choices: use a value from someone else, or find our own values, usually by trial and error.

There are a great many models available for various simulation programs. Probably the most freely available are those for spice, many of which can be downloaded from the semiconductor companies. Here, for example, is a typical spice model for a 1N4148 diode:[3]

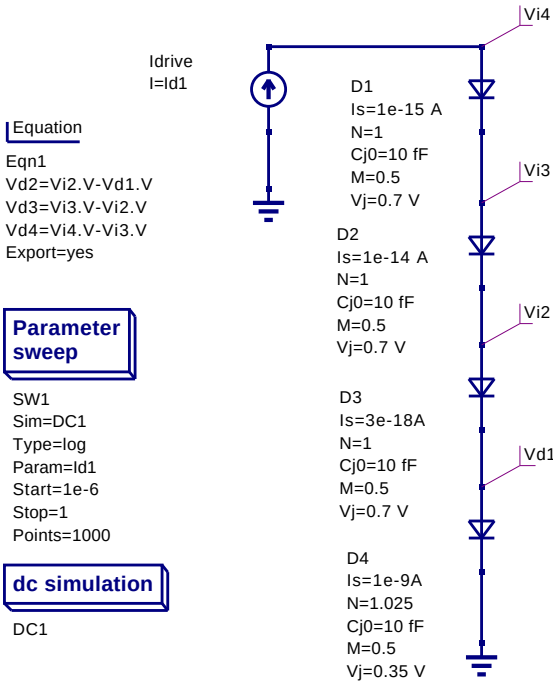
```
.model 1N4148  D(Is=0.1p Rs=16 CJO=2p Tt=12n Bv=100 Ibv=0.1p)
85-??-??   Original library
```

Any values not supplied are assumed to be the defaults.

The other way is to create your own device parameters, which is a bit like catching worms before you can go fishing. Insert values, plot the resulting characteristics, see how they compare with the published data sheet values, go back and adjust the values; continue until satisfied or exhausted.

Here, for example, is a circuit for quickly comparing the forward characteristics of diodes with different parameter values.

[3] I don't know where this came from, so I can't acknowledge the author. Most libraries are copyright, even if freely available.



And here is the plotted output...

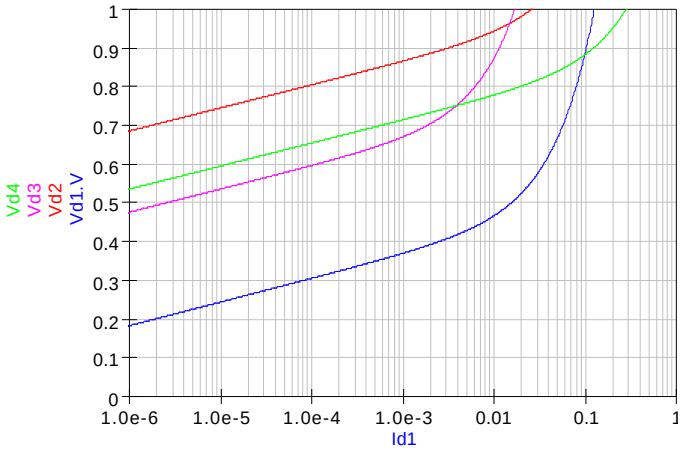


Figure 4.3: Diode Forward Voltage

The green and purple curves are typical of 1N4148 and 1N4448 devices; the others are medium and low-barrier Schottky devices. I have done a first pass compare with the data sheets, but I can't guarantee that these curves are any more than my best estimates.[4]

If you want to know more details of what each parameter does, there has been a great deal written over the years, particularly for spice, on the subject; a google search will quickly

[4]I'm assuming you are sick of screenshots by now, so I've just printed the schematic and display files from Qucs; you'll find the print item in the file menu, and if you ask it nicely it will print a postscript file.

reveal most of it. Qucs comes with a document which lists the details of its models, and, being open source, there is always the code itself.

Most of us end up taking a great deal on trust, and matching curves to data sheets as best we can. This is yet another instance of one of the fundamentals of engineering, the Duck Principle[5]: If you can't detect any difference between the behaviour of your model and the physical device, then they are, for engineering purposes, the same. Put it another way, when the difference between the model and the real device drops below the usual level of measurement uncertainty, it does matter any more.

In any case, component spreads in the real world tend to make the fine details of model inaccuracies somewhat academic, as we shall see when we model more complex devices.

[5]Usually expressed as: If it looks like a duck, walks like a duck, quacks like a duck and tastes like a duck, then, for all practical purposes, it is a duck.

5 Getting Started with Digital Circuit Simulation

5.1 Introduction

On 21 January 2006 Qucs 0.0.8 was released by the Qucs development team. This is the first version of the package to include digital circuit simulation based on VHDL. FreeHDL[1] being chosen as the VHDL engine. In the period following the release of Qucs 0.0.8 there has been considerable activity centred around finding and correcting a number of bugs in the Qucs digital simulation code. Many of these fixes are now included in the latest CVS code and will eventually form part of the next Qucs release. This tutorial note is an attempt on my part to communicate to other Qucs users a number of background ideas concerning the capabilities and limitations of the current state of Qucs VHDL simulation. Much of the information reported here was assembled by the author while assisting Michael Margraf to test and debug the VHDL code generated by Qucs. In the future, if there is enough interest in these notes, or indeed in Qucs VHDL simulation in general, I will update them as the Qucs digital simulation features are improved.

Qucs digital simulation follows a complex set of steps that are mostly transparent to the software user. In step one, a schematic representing a digital circuit under test is drawn. This schematic consists of an interconnected group of Qucs digital components, one or more user defined digital subcircuits (if required), and a copy of the digital simulation icon with the timing or truth table parameters set. In step two, the information recorded on a circuit schematic is converted into a text file containing VHDL statements. These describe the circuit components, their connection, and a testbench for simulating circuit performance. Next, FreeHDL is launched by Qucs to convert the VHDL code file into a C++ source program. This is compiled to form an executable machine code simulation of the original circuit. Finally, Qucs runs this program, collects signal data as digital signal events take place and displays signal waveforms as a function of time or digital data in a truth table format.

The VHDL code generated by Qucs 0.0.8 is limited in its scope by the following factors:

- Digital gates are described by data flow concurrent statements.
- Flip-flops and the digital signal generator are described by process statements.

[1] The FreeHDL Project, `http://www.freehdl.seul.org/`.

- Component connection wires (signals) can only be of type bit as defined in the standard VHDL library[2].
- Digital bus structures are not allowed in this release of the Qucs package.
- Digital subcircuits can be drawn as schematics and associated with a symbol in a similar fashion to analogue subcircuits.
- Digital subcircuit pins can have type in, out, inout or analog. Qucs treats pins of type analog the same as VHDL pin type inout.
- Once defined digital subcircuits may be placed and connected to other components on schematics.
- Multiple copies of the same digital subcircuit are allowed on a single schematic.
- Digital subcircuits may also be nested; nesting has been tested to a depth of four.

5.2 Simulating simple digital circuits

The most basic form of digital circuit that can be simulated is one consisting entirely of Qucs predefined digital components drawn on a schematic having only one level of design hierarchy. The truth table for a simple combinational circuit of this type is shown in Table 8.1.

Output F can be expressed in sum of products Boolean form as

$$F = \overline{A}.\overline{B}.C + \overline{A}.B.\overline{C} + A.\overline{B}.C + A.B.\overline{C}$$

[2]Signal type bit only defines logic signals '0' and '1'. Care must be taken to ensure that signal contention does not occur during simulation because the resulting logic state cannot be modelled with type bit. Signal contention can happen when two or more digital devices attempt to drive the same wire with logic '0' and logic '1' signals at the same time. Moreover, it is not possible to simulate the performance of tristate devices using VHDL signal type bit.

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Table 5.1: Truth table for a logic circuit with inputs A, B, C and output F.

On minimisation, using Boolean algebra or a Karnaugh map, output F becomes

$$F = A.C + B.\overline{C}$$

The schematic for example 1 is illustrated in Fig. 5.1. This diagram was constructed using the same techniques employed for drawing analogue schematics.

5.2.1 Notes on drawing digital schematics

- The only predefined Qucs components that can be used to draw a digital circuit schematic are (1) the digital components listed in the digital components icon window, (2) the ground symbol, and (3) the digital simulation icon.
- A useful tip when drawing digital schematics is to adopt the matrix approach shown in Fig. 5.1. Input signals flow from top to bottom of the schematic and output signals are positioned on the right-hand side of a horizontal line. This makes checking the circuit schematic for errors much easier than the case where diagrams have wires connecting components in an unstructured way.
- Input and output wires (signals) should be given names consistant with the circuit being simulated, A, B, C and F in Fig. 5.1. If the signal wires are not named by the user, Qucs will allocate them different arbitrary names. This can make identification and selection of signals for display on an output waveform graph, and indeed checking for errors in a large circuit, much more difficult than it need be.
- Notice in Fig. 5.1 the international symbols for the logic gates are shown on the schematic.

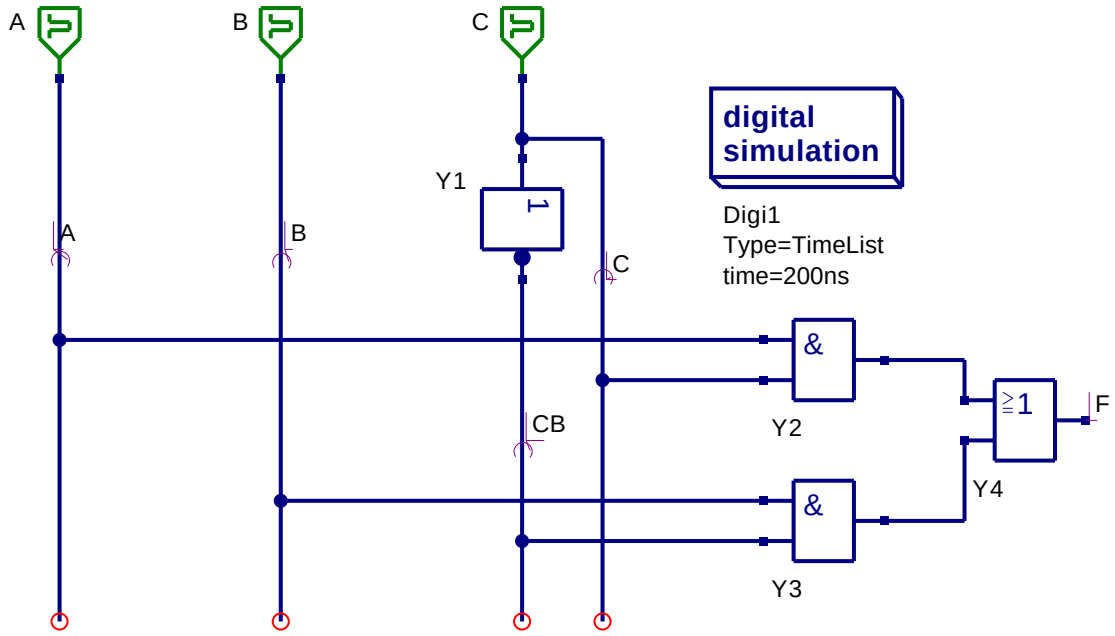


Figure 5.1: Qucs schematic for minimised logic function F.

5.3 VHDL code generated by Qucs

Clicking the Qucs Simulate menu button (or pressing key F2) starts the simulation process. At an early phase in this process Qucs writes a text file to disk that contains the VHDL code for the circuit being simulated. This file can be displayed by clicking on the ***show last netlist*** drop down menu or by pressing key F6. The VHDL code produced by Qucs for the circuit shown in Fig. 5.1 is presented in Table 6.1.

Signals identified by nnnet0 and nnnet1 in Table 6.1 have been allocated these names by Qucs; nnnet0 and nnnet1 are internal signal nets that are not named on the circuit schematic shown in Fig. 5.1. Fig. 5.2 illustrates the starting section of a typical Qucs digital functional waveform plot. This style of plot illustrates signal events without component delays. If required, signal delays can be specified for individual gates and other components (from the component ***edit properties*** menu). The VHDL code generated for components with delays will then reflect such changes, for example adding a 10 ns delay to signal CB in Table 6.1 generates VHDL code

```
CB <= not C after 10 ns;
```

Readers will probably have observed that the Qucs version number referred to in Table 6.1 VHDL listing is 0.0.9. This is the current CVS development version number. Qucs 0.0.9 includes a number of important bug fixes. The remainder of these notes assume readers have downloaded, and recompiled, the latest CVS code from Sourceforge.net[3].

[3]Please note, Qucs Linux release 0.0.8 will normally simulate single hierarchy digital circuits without

```
-- Qucs 0.0.9  tut1_ex1.sch
entity TestBench is
end entity;
use work.all;

architecture Arch_TestBench of TestBench is
signal CB, A,  B,  F, C,
       nnnet0,
       nnnet1 : bit;
begin
  nnnet0 <= C and A;
  nnnet1 <= CB and B;
  CB <= not C;

  A:process
  begin
    A <= '0';  wait for 40 ns;
    A <= '1';  wait for 40 ns;
  end process;


  B:process
  begin
    B <= '0';  wait for 20 ns;
    B <= '1';  wait for 20 ns;
  end process;

  F <= nnnet1 or nnnet0;

  C:process
  begin
    C <= '0';  wait for 10 ns;
    C <= '1';  wait for 10 ns;
  end process;

end architecture;
```

Table 5.2: VHDL code for the circuit shown in Fig. 5.1.

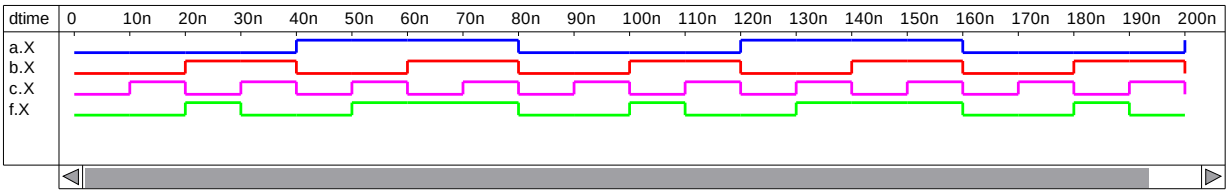


Figure 5.2: Digital functional waveforms for the circuit shown in Fig. 5.1.

5.4 Truth tables

Truth tables are one of the most fundamental and convenient ways of displaying digital circuit data. Qucs has a built-in facility that allows a truth table to be generated from a schematic drawing. This feature is particularly useful when checking minimised logic designs for errors. Lets consider a simple but instructive example: A logic circuit has four binary inputs A, B, C, and D, and one output P. Output P is logic '1' when inputs ABCD are numbers in the decimal sequence 3, 5, 7, 11 and 13. In Boolean sum of product form

$$P = \overline{A}.\overline{B}.C.D + \overline{A}.B.\overline{C}.D + \overline{A}.B.C.D + A.\overline{B}.C.D + A.B.\overline{C}.D$$

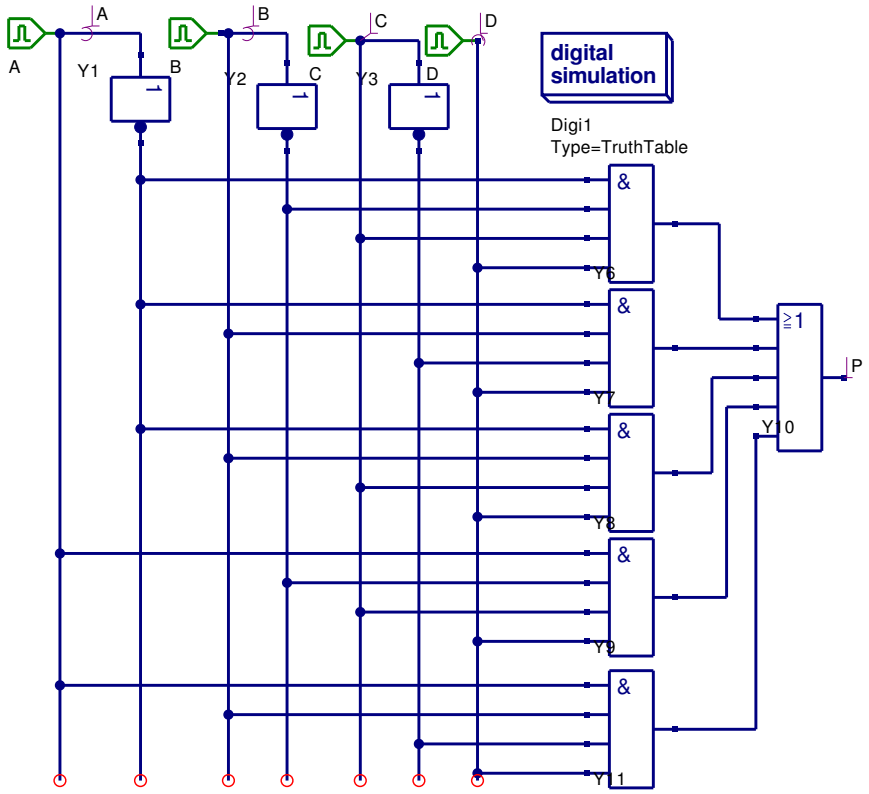
This simplifies to

$$P = D.(A.B + B \oplus C)$$

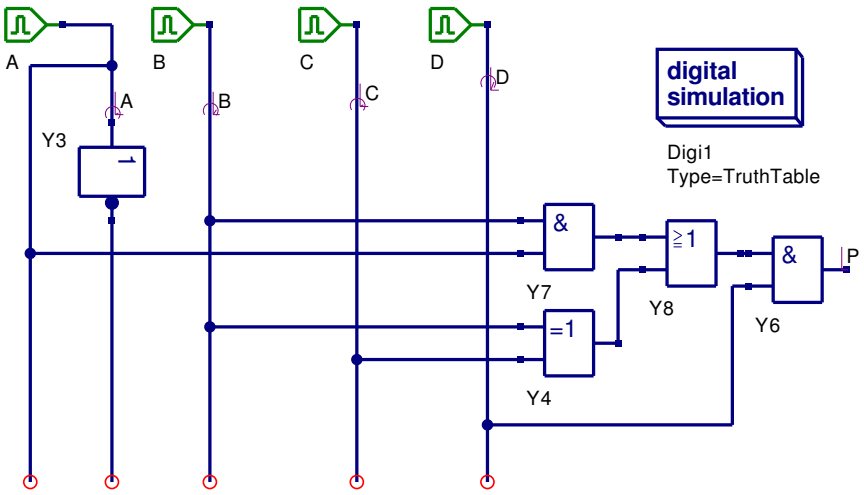
The schematic for the sum of products equation for P is shown in Fig. 5.3(a). Similarly Fig. 5.3(b) presents the schematic for a minimised P equation. Setting the digital simulation type to TruthTable, rather than TimeList, causes Qucs on pressing key F2, to generate a truth table based on the information provided on a circuit schematic. The number of truth table inputs, and indeed outputs, correspond to the number of input generators and the number of named outputs. Truth tables for both schematics are given in Table 5.3(a) and 5.3(b). Comparing these two tables clearly indicates that they are not identical and moreover confirms that the minimised solution is not correct. Reworking the minimisation procedure points to the error being a missing signal inversion. The correct Boolean equation for P is

$$P = D.(\overline{A}.B + B \oplus C)$$

error. However, Qucs 0.0.8 does fail at the VHDL to C++ conversion phase if a schematic includes more than one ground symbol.



5.3(a): Schematic diagram for sum of products equation P



5.3(b): Schematic diagram for minimised equation P

5.3(a): Truth table for sum of products equation P

	a.X	b.X	c.X	d.X	p.X
00000	0	0	0	0	0
00001	0	0	0	1	0
00010	0	0	1	0	0
00011	0	0	1	1	1
00100	0	1	0	0	0
00101	0	1	0	1	1
00110	0	1	1	0	0
00111	0	1	1	1	1
01000	1	0	0	0	0
01001	1	0	0	1	0
01010	1	0	1	0	0
01011	1	0	1	1	1
01100	1	1	0	0	0
01101	1	1	0	1	1
01110	1	1	1	0	0
01111	1	1	1	1	0

5.3(b): Truth table for minimised equation P

	a.X	b.X	c.X	d.X	p.X
00000	0	0	0	0	0
00001	0	0	0	1	0
00010	0	0	1	0	0
00011	0	0	1	1	1
00100	0	1	0	0	0
00101	0	1	0	1	1
00110	0	1	1	0	0
00111	0	1	1	1	0
01000	1	0	0	0	0
01001	1	0	0	1	0
01010	1	0	1	0	0
01011	1	0	1	1	1
01100	1	1	0	0	0
01101	1	1	0	1	1
01110	1	1	1	0	0
01111	1	1	1	1	1

5.5 Digital subcircuits

Although it is possible to draw complex schematic diagrams using only the predefined digital components supplied with Qucs, this technique can be extremely tedious, and is of course, prone to error. When drawing large schematics we require a design procedure that naturally subdivides groups of digital components into self contained units. These units can then be treated in the same way as basic digital components when placing and connecting them on a schematic drawing. In the world of analogue and digital circuit design such units are often called subcircuits.[4] A subcircuit is defined by three major attributes plus a number of other properties. The major attributes are, firstly a digital circuit that defines circuit function, secondly a circuit symbol that depicts a circuit in a higher level of a design hierarchy, and thirdly the subcircuit input/output pins shown on the subcircuit symbol. Other properties include for example, signal path delays. The process for generating digital subcircuits is identical to that used for analogue subcircuits. It is best demonstrated by considering an example. Figure 5.4 shows the schematic for a four input combinational circuit.

After drawing a subcircuit schematic, input and output[5] pins are attached to signal ports. Input port pins of type in are shown on circuit diagrams as a green symbol, signals W, X, Y, and Z, in Fig. 5.4. Ouput port pins of type out are coloured red, signal G in Fig. 5.4. Signal flow through a port is indicated by the direction of the port symbol arrow head. Input/output signals, and any other signals that need to be easily identified, are also named. Once the subcircuit schematic is complete, pressing key F3 causes Qucs to generate a subcircuit symbol. The drawing tools listed as icons in the Qucs paintings window can be used to edit Qucs generated subcircuit symbols. The input/output port pins on a subcircuit symbol have the same type and name as those on the original subcircuit schematic. Fig. 5.5 shows the finished symbol for subcircuit COMB1. In these notes, symbol outlines are shown drawn in accordance with the international code for logic symbols[6]. To test our new subcircuit we place it's symbol on a blank drawing sheet and apply test signals to the input pins and observe the signals at the output pin. Fig. 5.6 shows a typical test circuit. Subcircuit Gen4bit generates a 4 bit test pattern synchronised to the input of a digital clock. The specification for Gen4bit is given in the next section of these notes[7]. The test pattern waveform and output signal G are shown plotted as a function of time in Fig. 5.7.

[4]The circuit simulator SPICE is a well known example of a widely used CAD program that makes extensive use of subcircuits in circuit design.

[5]Qucs 0.0.8 has a bug which causes a VHDL compile error when subcircuit pins are specified as type out. A work around for this bug is to specify subcircuit output pins as type analog. The Qucs routines that generate the circuit VHDL code convert pin type analog into VHDL type inout. FreeHDL is then able to compile the generated VHDL code without error. This bug has been corrected in Qucs 0.0.9.

[6]Ian, Kampel, A practical introduction to the new logic symbols, Butterworths, 1985, ISBN 0-408-01461-X.

[7]Subcircuit Gen4bit includes other nested subcircuits. Qucs 0.0.8 has a bug that causes VHDL compile errors with some configurations of nested subcircuits. This has been fixed in version 0.0.9.

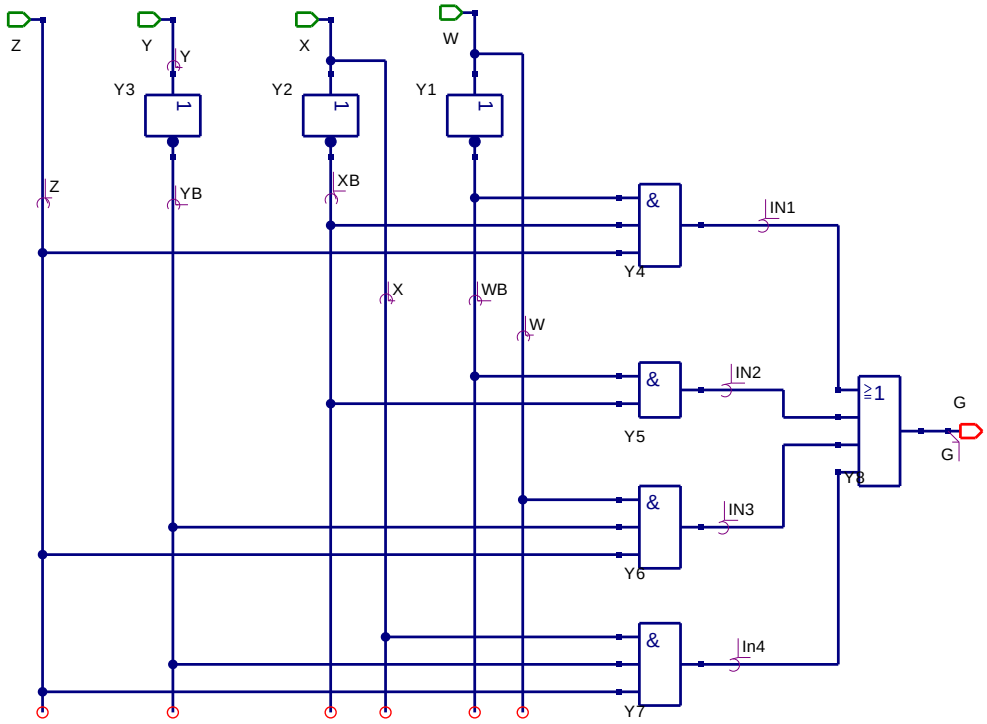


Figure 5.4: Combinational logic circuit with inputs W, X, Y, Z, and output G.

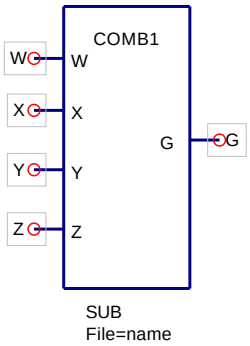


Figure 5.5: Qucs symbol for a logic circuit with inputs W, X, Y, Z, and output G.

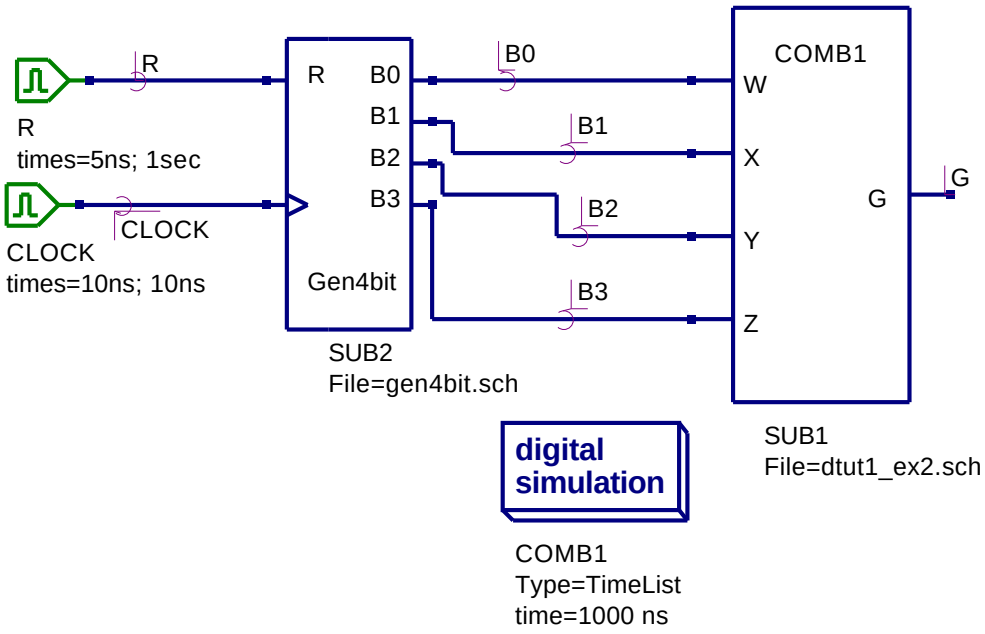


Figure 5.6: Test schematic for a logic circuit with inputs W, X, Y, Z, and output G.

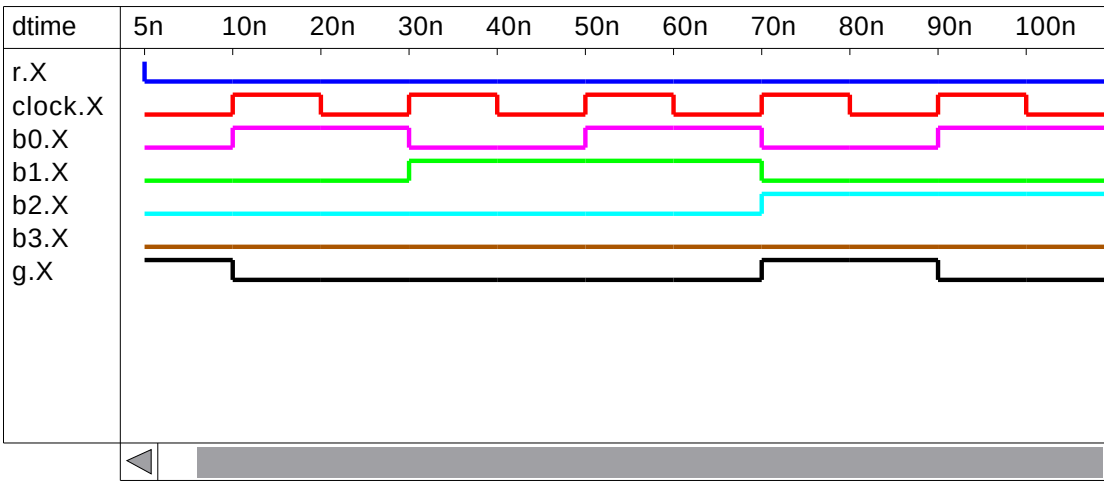
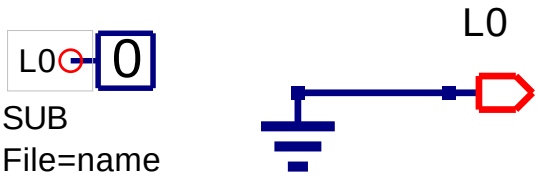


Figure 5.7: Digital functional waveforms for a logic circuit with inputs W, X, Y, Z, and output G.

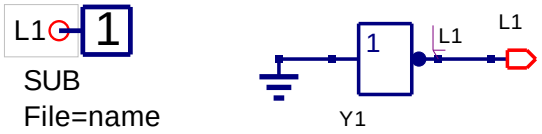
5.6 Building a digital component library

The Qucs graphical user interface includes good project handling features. Combining these features with the Qucs subcircuit capabilities provides all the tools required for the development of a library of common digital components. Such a library can be stored in a master project and the individual component files imported into other projects when required. Here are a few components that I developed during a recent series of tests aimed at detecting bugs in the VHDL code generated by Qucs.

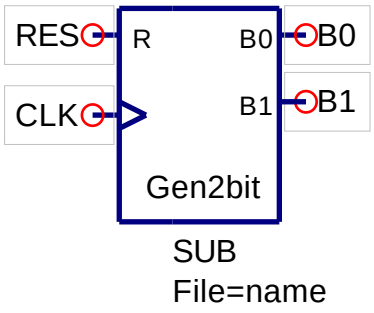
5.6.1 Logic zero

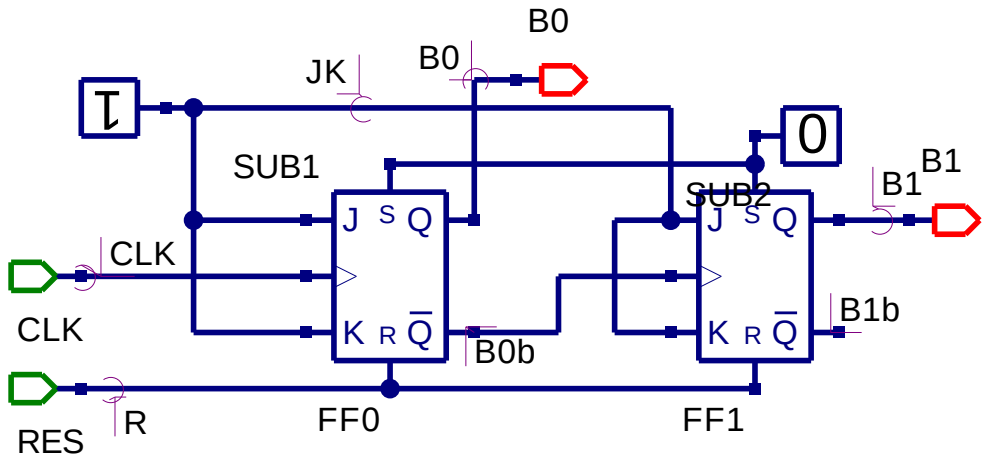


5.6.2 Logic one

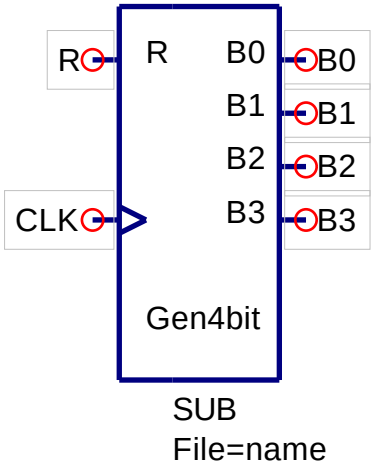


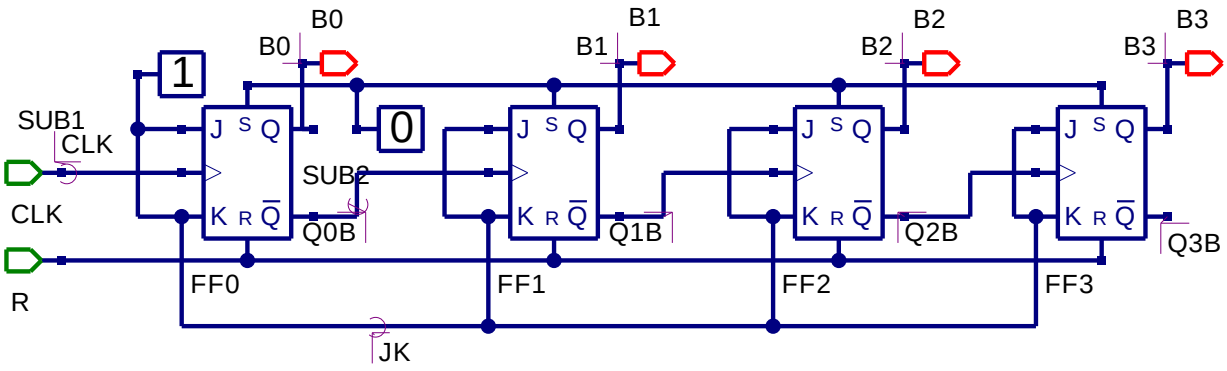
5.6.3 G2bit - 2 bit pattern generator





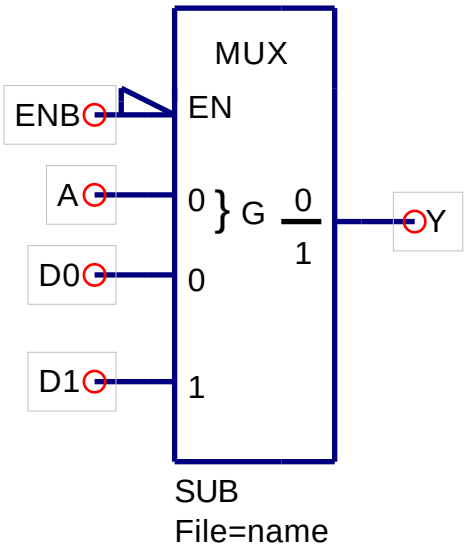
5.6.4 G4bit - 4 bit pattern generator

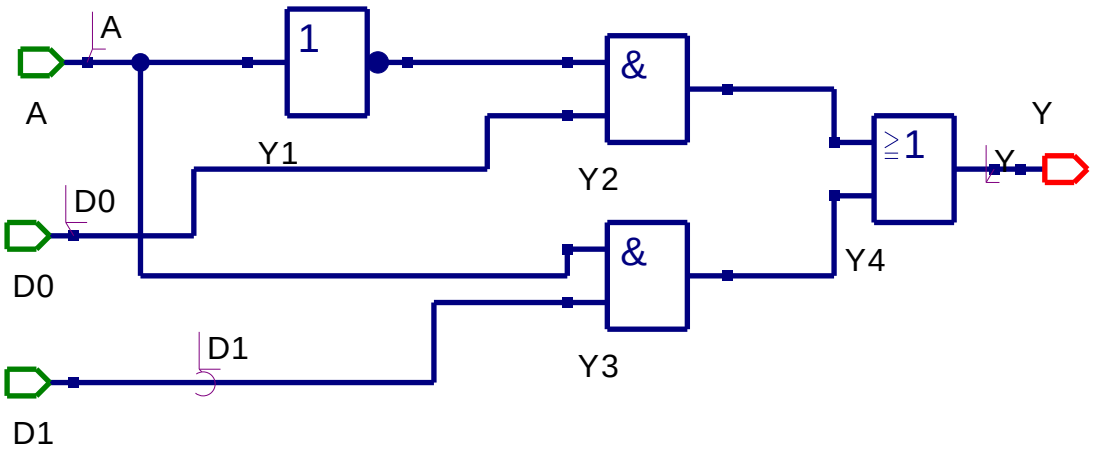




5.6.5 MUX2to1 - 2 input to 1 output multiplexer

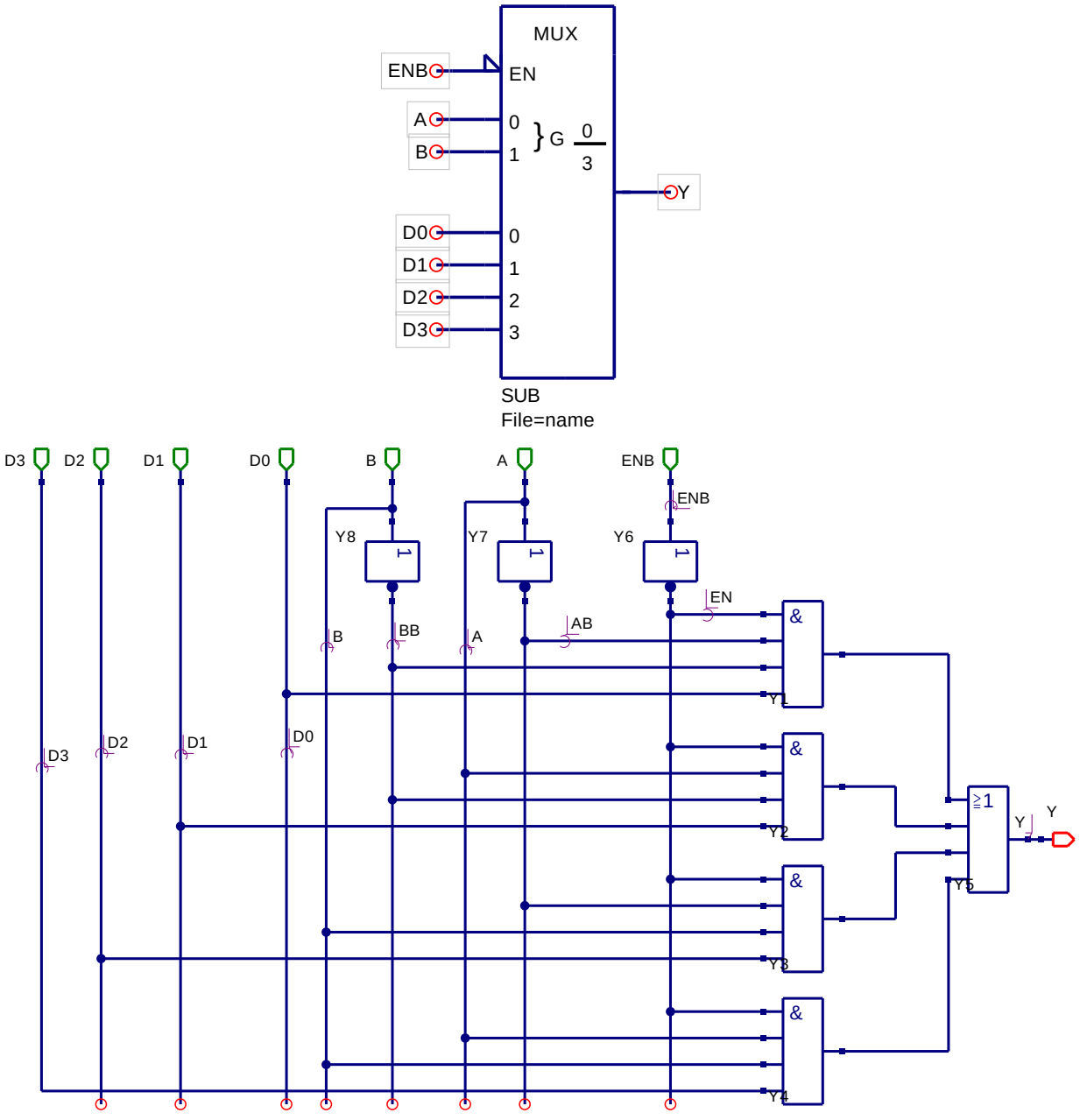
EN	A	Y
1	X	L
0	0	D0
0	1	D1



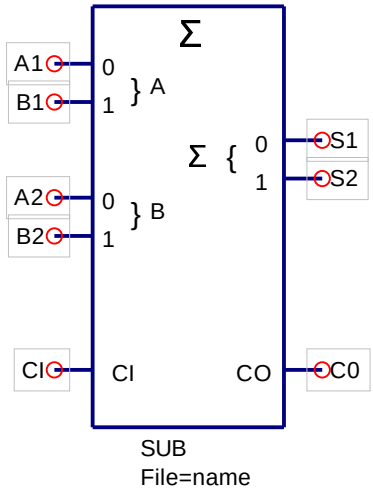


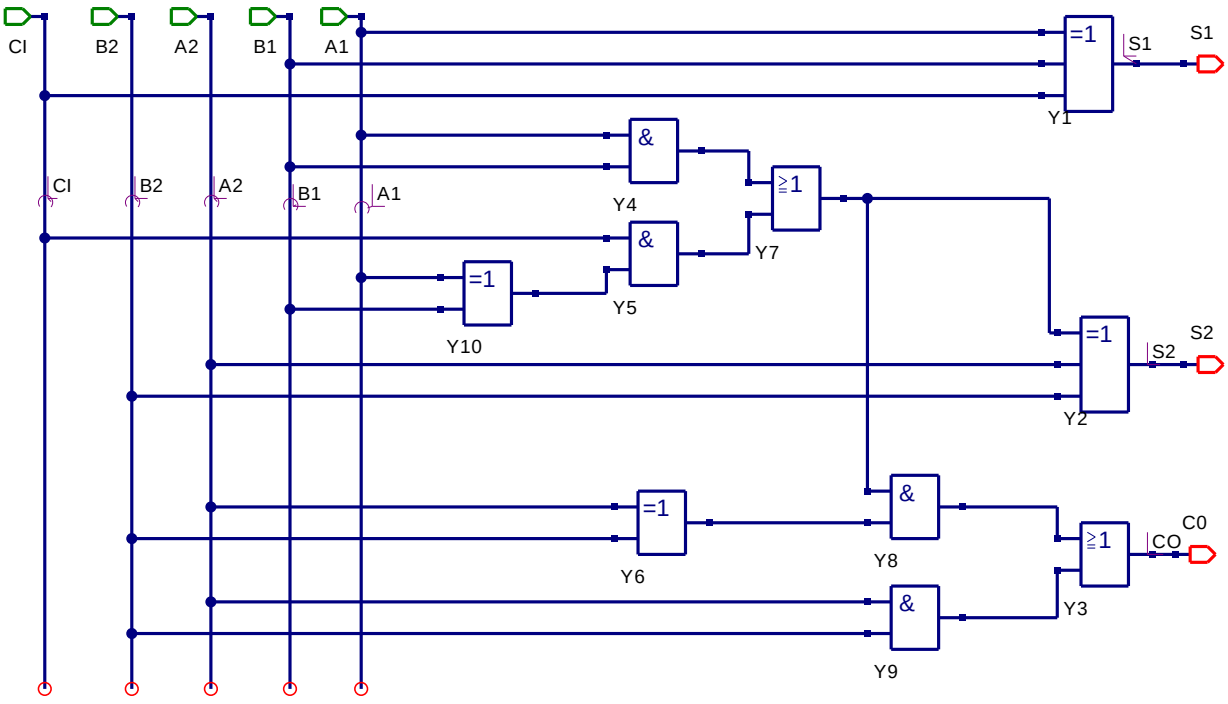
5.6.6 MUX4to1 - 4 input to 1 multiplexer

B	A	EN	Y
X	X	1	0
0	0	0	D0
0	1	0	D1
1	0	0	D2
1	1	0	D3



5.6.7 2 bit adder





5.7 Subcircuit VHDL code generated by Qucs

Qucs generates a separate entity-architecture model for each subcircuit. These component definitions are compiled into the work library by FreeHDL. Here is the VHDL code from two of the previous examples.

5.7.1 Gen2bit

```
entity Sub_gen2bit is
  port (CLK: in bit;
        R: in bit;
        nnout_B0: out bit;
        nnout_B1: out bit);
end entity;
use work.all;
architecture Arch_Sub_gen2bit of Sub_gen2bit is
  signal B0b,
         B1b,
         JK,
         nnnet0,
         B0,
         B1 : bit;
begin
  FF0 : process (nnnet0, R, CLK)
  begin
    if (R='1') then  B0 <= '0';
    elsif (nnnet0='1') then  B0 <= '1';
    elsif (CLK='1' and CLK'event) then
      B0 <= (JK and not B0) or (not JK and B0);
    end if;
  end process;
  B0b <= not B0;

  FF1 : process (nnnet0, R, B0b)
  begin
    if (R='1') then  B1 <= '0';
    elsif (nnnet0='1') then  B1 <= '1';
    elsif (B0b='1' and B0b'event) then
      B1 <= (JK and not B1) or (not JK and B1);
    end if;
  end process;
  B1b <= not B1;

  SUB2: entity Sub_logic_zero port map (nnnet0);
  nnout_B0 <= B0 or '0';
  nnout_B1 <= B1 or '0';
  SUB1: entity Sub_Logic_one port map (JK);
end architecture;
```

5.7.2 2 bit adder

```
entity Sub_fadd_2bit is
  port (A1: in bit;
        B1: in bit;
```

```
        A2: in bit;
        B2: in bit;
        CI: in bit;
        nnout_S1: out bit;
        nnout_S2: out bit;
        nnout_CO: out bit);
end entity;
use work.all;
architecture Arch_Sub_fadd_2bit of Sub_fadd_2bit is
  signal nnnet0,
         nnnet1,
         nnnet2,
         nnnet3,
         nnnet4,
         nnnet5,
         nnnet6,
         S2,
         CO,
         S1 : bit;
begin
  S1 <= CI xor B1 xor A1;
  nnnet0 <= B2 xor A2;
  nnnet1 <= nnnet0 and nnnet2;
  nnnet3 <= B2 and A2;
  nnnet2 <= nnnet4 or nnnet5;
  nnnet4 <= nnnet6 and CI;
  nnnet5 <= B1 and A1;
  S2 <= B2 xor A2 xor nnnet2;
  CO <= nnnet3 or nnnet1;
  nnnet6 <= B1 xor A1;
  nnout_S2 <= S2 or '0';
  nnout_CO <= CO or '0';
  nnout_S1 <= S1 or '0';
end architecture;
```

5.7.3 Notes on subcircuit VHDL generation

- Qucs predefined digital components generate concurrent data flow signal statements or process statements.

- Previously defined subcircuit symbols generate VHDL port map statements.

- Type out entity port signals are prevented from being read as input signals by masking each output signal using the logic function **signal-name OR '0'**.[8]

[8] Attempting to read entity port signals of type out results in a VHDL compile error.

- A VHDL

  ```
  use work.all;
  ```

  statement is included before each subcircuit architecture definition to ensure that FreeHDL can find any nested subcircuits [9].

- The complete VHDL code file for a digital design is composed from an outer test bench entity-architecture model plus entity-architecture models for each subcircuit specified in the design,

5.8 Subcircuit nesting: A more complex design example

In theory there is no limit to the depth of subcircuit nesting allowed by Qucs. In practice most digital circuit schematics can be constructed with a maximum of four or five levels of design hierarchy. Figure 5.8 shows an example that was used to test Qucs subcircuit nesting performance. The design is a simple RTL function that uses a multiplexer to transfer data from one of two input registers to a single output register. The next section of these notes outlines in detail the specification of the subcircuits needed to build the RTL design. A set of sample simulation waveforms showing the register transfer operation are illustrated in Fig. 5.9.

[9] Strictly speaking it should not be necessary to specifically state the use of the work library as this library is normally visible at all times when compiling entity-architecture models. However, at this stage in the development of FreeHDL it does appear that it is necessary when using the default FreeHDL VHDL library mapping.

5.8.1 4 bit RTL design

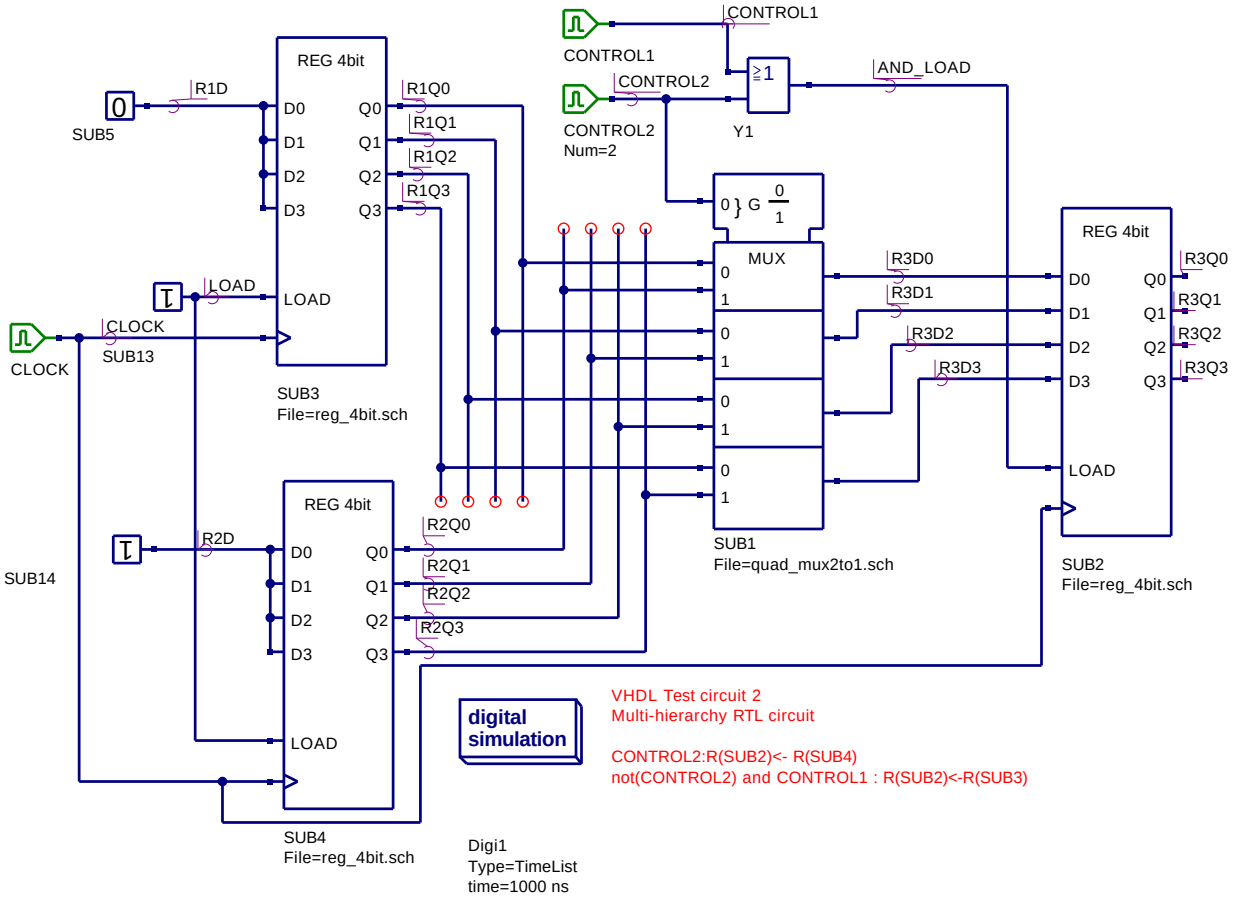
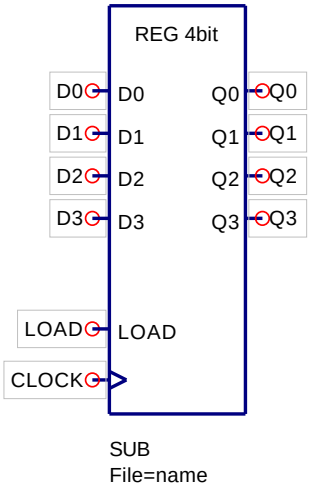
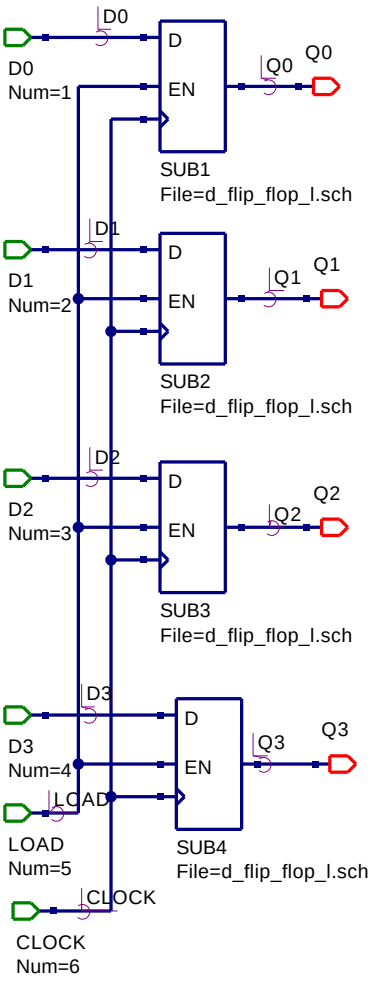


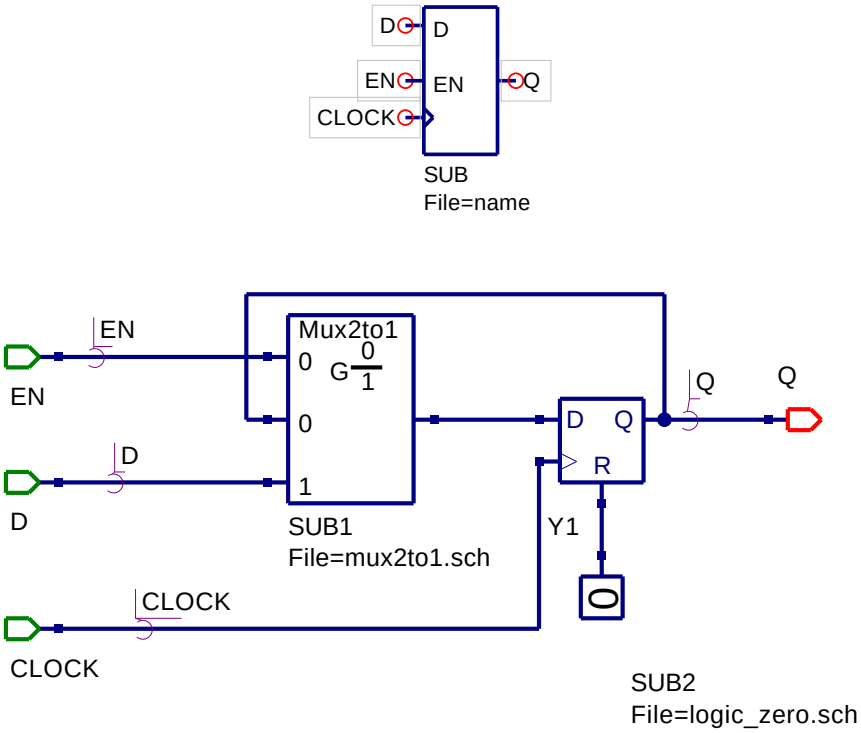
Figure 5.8: Top level schematic.

Reg4bit

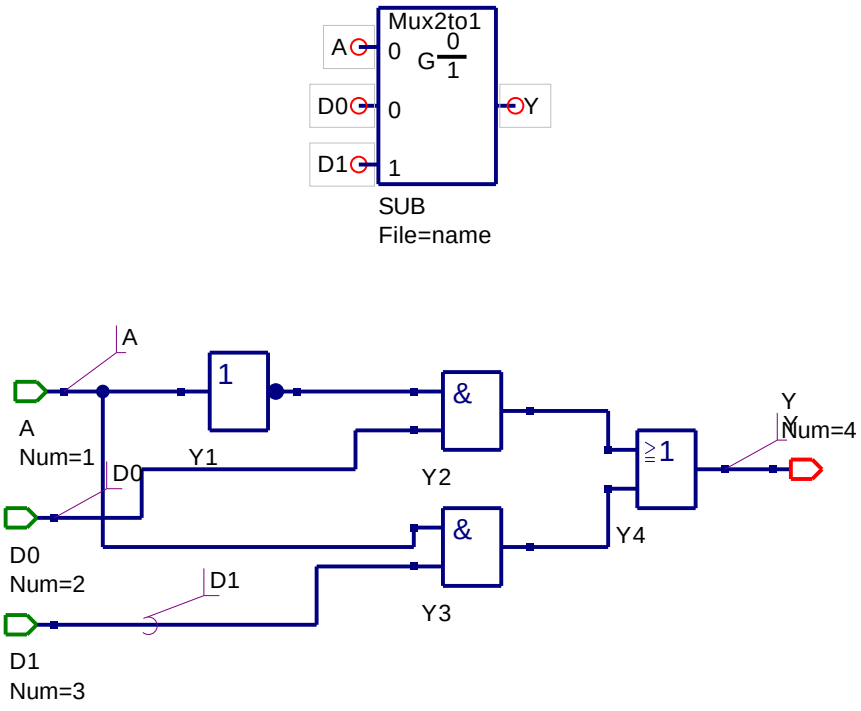

REG 4bit
D0
D1
D2
D3
LOAD
CLOCK
Q0
Q1
Q2
Q3
SUB
File=name


D0
Num=1
D1
Num=2
D2
Num=3
D3
Num=4
LOAD
Num=5
CLOCK
Num=6
D
EN
Q0
Q1
Q2
Q3
SUB1
File=d_flip_flop_l.sch
SUB2
File=d_flip_flop_l.sch
SUB3
File=d_flip_flop_l.sch
SUB4
File=d_flip_flop_l.sch

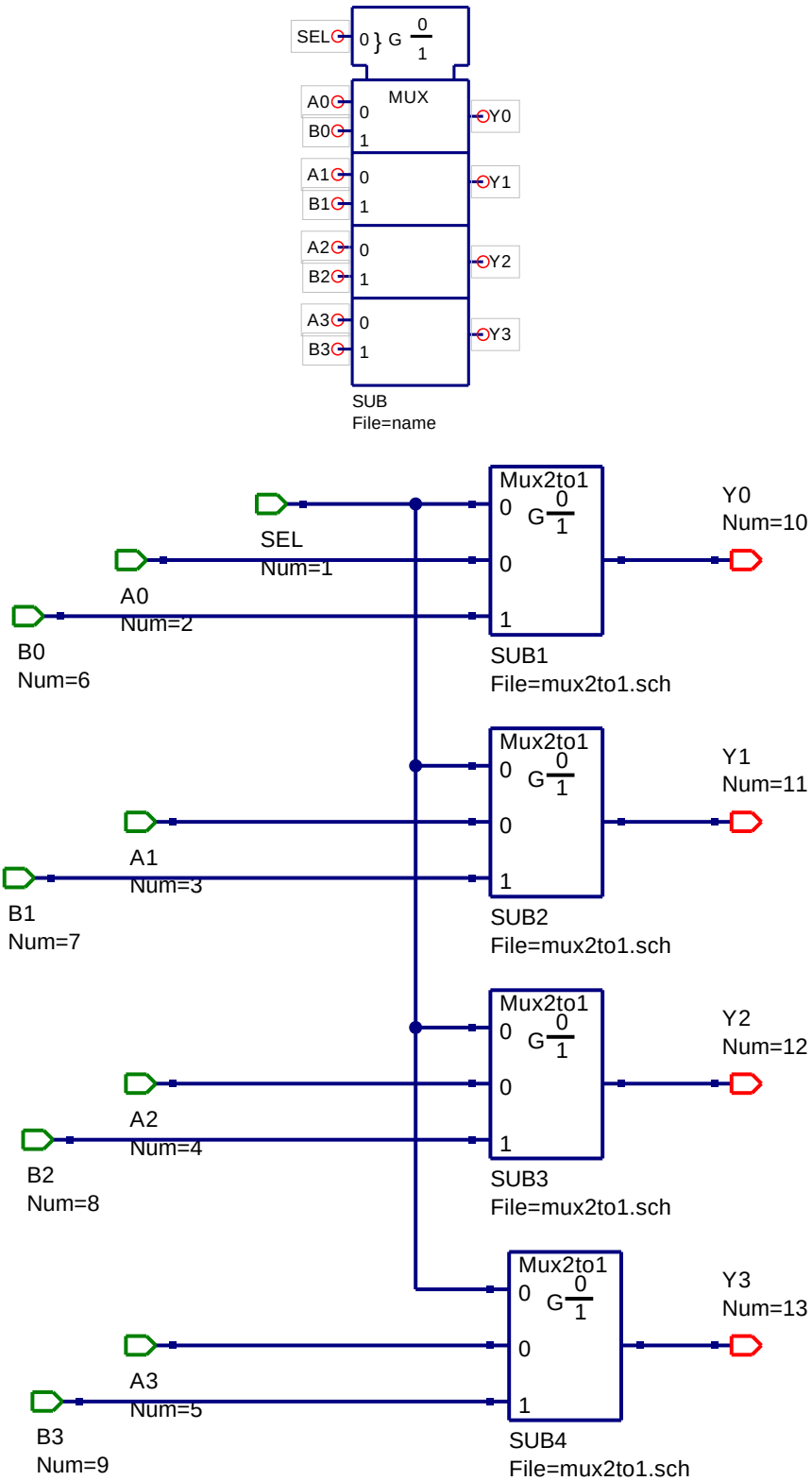
D flip-flop with load enable



Mux2to1



QuadMux


SEL
0
G
0
1
MUX
A0
B0
Y0
A1
B1
Y1
A2
B2
Y2
A3
B3
Y3
SUB
File=name
SEL
Num=1
A0
Num=2
B0
Num=6
Mux2to1
SUB1
File=mux2to1.sch
Y0
Num=10
A1
Num=3
B1
Num=7
SUB2
File=mux2to1.sch
Y1
Num=11
A2
Num=4
B2
Num=8
SUB3
File=mux2to1.sch
Y2
Num=12
A3
Num=5
B3
Num=9
SUB4
File=mux2to1.sch
Y3
Num=13

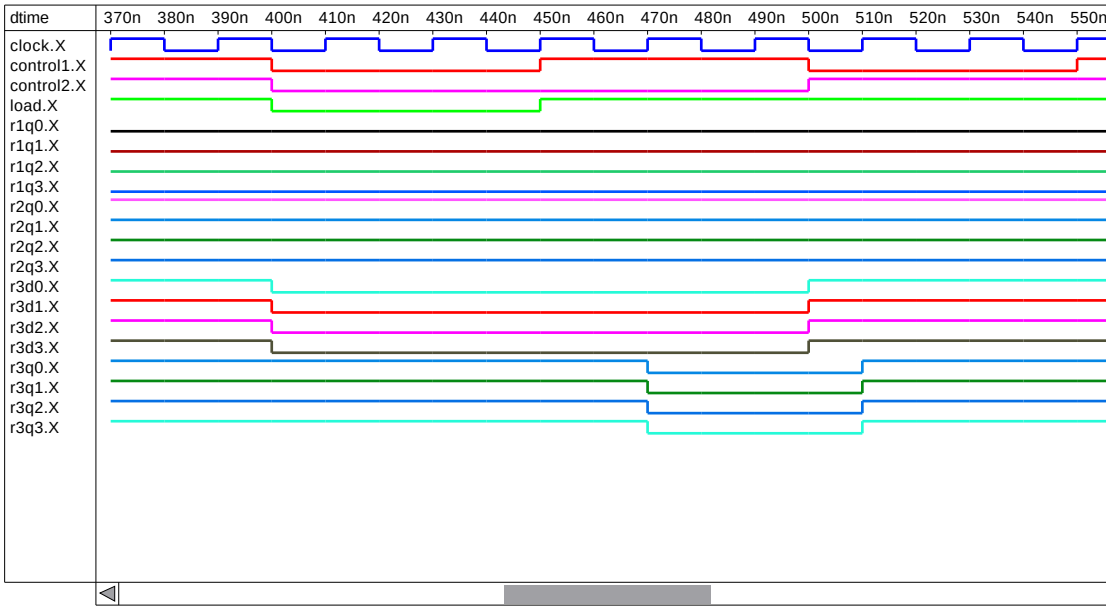


Figure 5.9: Sample simulation waveforms for RTL design.

5.9 Update number one: May 2006

Although it is only a short time since the first version of these digital tutorial notes was posted on the Qucs Sourceforge Web site, much has happened in the world of Qucs digital simulation. Bugs in the Qucs code have been found, and fixed, and a range of new features added to the software. These expand the power of Qucs digital simulation and give users a glimpse of how the package will evolve in the future. The purpose of these notes is firstly to update readers as to the changes to Qucs digital simulation and secondly to explain how to use the new Qucs features. Please note however, they are not intended to teach readers how to program using VHDL.[10]

5.9.1 Bugs, corrections and small changes to the Qucs digital simulation code

All the bugs reported in the first version of these notes have been corrected in the latest Qucs CVS code. These corrections are, of course, also included in Qucs release 0.0.9. During testing a number of other annoying, but significant, bugs have also been found and eliminated, these include

- Multiple input gates (three or more inputs) of types nand and nor failed at the FreeHDL compile stage due to an error in the VHDL code generated by Qucs.

[10] A good introduction to the VHDL language and it's application in digital system design can be found in ***Digital System Design using VHDL*** by Charles H. Roth, Jr, PWS Publishing Company, 1997, ISBN 0-534-95099-X.

- Signals names and, for example, component names constructed from a single letter that was an abbreviation for a physical unit failed to compile.
- Changing digital component time delays caused component connections on a schematic to be removed.
- GUI problems caused by errors in the symbol rotation and mirror code.
- Qucsconv code conversion errors caused the Qucs digital simulation cycle to fail before plotting TimeList waveforms.

A number of changes to either the VHDL code generated by Qucs or the schematic capture GUI have been introduced, these include

- The VHDL code generated by Qucs for the ground symbol has been changed from

  ```
  gnd <= gnd and '0';
  ```

  to

  ```
  gnd <= '0';
  ```

- The symbol for digital inout ports has been changed from the analogue pin symbol to one that consists of the digital in and out pins drawn back-to-back. This reflects the bidirectional status of an inout port.

A more complete list of all the bug corrections and other program modifications can be found in the Qucs change log files.

5.9.2 New digital simulation features

The flow diagram illustrated in Fig. 5.10 shows a number of different simulation routes for a digital circuit under test. The Qucs digital simulation facilities have been improved to include direct simulation of VHDL testbench code and the simulation of circuit schematics that include digital components specified by VHDL entity-architecture models. The various combinations that users can adopt for Qucs digital circuit entry are as follows:

1. Schematic circuit entry using predefined digital component symbols, subcircuits generated using the same symbols and a copy of the digital simulation icon; this is the approach described in the first version of these tutorial notes.
2. Circuit entry identical to 1 plus symbols for digital components specified by VHDL entity-architecture models.
3. Circuit entry using the Qucs VHDL code editor. The text entered describes both the circuit under test and the test vectors needed to drive the circuit inputs during simulation.

Once the circuit under test has been entered into Qucs, clicking the Simulate menu button, or pressing key F2, starts the Qucs digital simulation process.

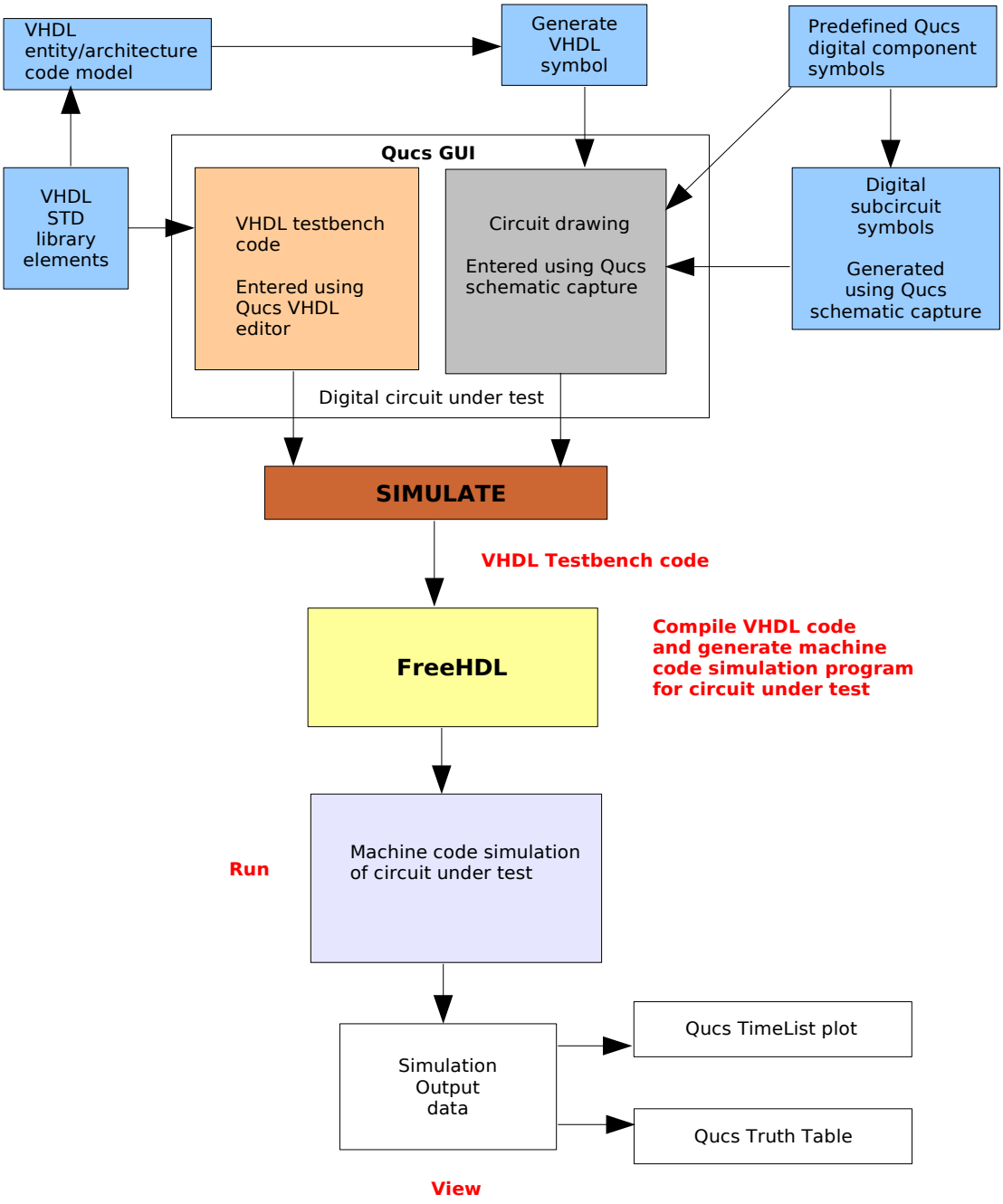


Figure 5.10: Flow diagram of Qucs digital simulation routes.

5.9.3 Limitations

Before describing the new digital simulation features it is important that readers understand the limitations that are inherent in the various digital simulation routes described in the last section and illustrated in the flow diagram shown in Fig. 5.10. Qucs schematic capture allows users to draw circuits consisting of predefined component symbols and subcircuit symbols. At this stage in the development of the GUI digital signals must be of type bit (as defined in the VHDL standard library - library STD in the FreeHDL package) where individual signals flow through a single wire. Qucs schematic drawing bus structures of VHDL type bit-vector, for example, have not been implemented yet. This implies that the device symbol port pins must represent single signals. Similarly the nets connecting pins on more than one device can only be single signal nets and not bus structures. It is anticipated that this will change in a future Qucs release.

Although the current release of FreeHDL is 0.0.1 the package implements a substantial subset of the entire VHDL language[11]. The major features not supported by release 0.0.1 are:

- Shared variables.
- The following attributes; transaction, quiet, stable and delayed.
- User defined attributes.
- Groups.
- Guarded signal assignments.
- Currently drivers cannot be switched off.

The Qucs TimeList plotting program uses signal data output by the machine code simulation program generated by the FreeHDL package[12]. A current limitation of the TimeList plotting program is that it can only display signals of type bit. Bus signal waveforms cannot be displayed.

Given the above limitations it is therefore possible to write VHDL code that can be compiled by FreeHDL but will cause problems at either the schematic drawing or output waveform plotting stages in the Qucs simulation cycle. As Qucs develops it is expected that these limitations will be removed. On the subject of limitations one final point to note: FreeHDL can simulate circuits described by the data types and other features found in the

[11] A complete description of the 1987 and 1993 specifications of the VHDL language can be found in The Designer's Guide to VHDL by Peter J Ashenden, second edition 2002, Morgan Kaufmann Publishers, ISBN 1-55860-674-2.

[12] The machine code simulation program outputs signal data in VCD format. This is then converted to the Qucs TimeList data format by the qucsconv utility program.

```
IEEE.std_logic_1164
```

library and other predefined libraries. However, at this stage in the development of the Qucs software only the VHDL standard library may be used, implying that data type bit must be used to represent logic signals.

5.9.4 Using the Qucs VHDL editor

Qucs release 0.0.9 includes a VHDL text editor[13] that has all the usual edit features plus colour coding of the various VHDL language statements. One unusual feature of this editor is a zoom control that allows the text size to be increased or decreased in a similar way to the schematic drawing zoom. The VHDL editor is included in the Qucs package for two primary purposes, firstly for purely text file VHDL simulation[14] and secondly for the development of VHDL entity-architecture models that can be linked to schematic capture symbols. The latter increases significantly the capabilities of the Qucs software in that it allows libraries of hand-crafted device models to be constructed. These new library devices will, given support by the general Qucs user community, greatly expand the potential use of the Qucs package. In this section the use of the VHDL text editor is demonstrated through a series of digital circuit simulation examples. The included VHDL listings indicate typical Qucs use of a number of the basic VHDL data types. The text also outlines any limitations imposed by Qucs.

- Example 1: A sum of products (SOP) combinational digital circuit.

  The Boolean equation[15] for a SOP combinational circuit is:

$$f = \overline{W}.X.\overline{Y}.\overline{Z} + \overline{W}.\overline{X}.\overline{Y}.\overline{Z} + W.\overline{Y}.\overline{Z} + W.X.Y.Z$$

  The VHDL code for a structural model of this combinational circuit and its associated testbench is given in the following listing.

```
-- Qucs VHDL editor example 1
--
entity test_vector is  -- Test vector generator.
        port( z, y, x, w : out bit
              );
end entity test_vector;
--
architecture behavioural of test_vector is
```

[13] To launch the new VHDL editor click on the second icon from the left on the Qucs toolbar. It can also be activated using the key sequence Ctrl+Shift+V.

[14] This is still the preferred method amongst many experienced users of VHDL. However, the circuit schematic drawing approach does seem to be growing in popularity.

[15] The Boolean equation for function f has not been minimised. It is in a form derived directly from a truth table and is introduced purely as an example to demonstrate the use of the Qucs VHDL editor.

```
begin
pz : process is
        begin
                z <= '0' ; wait for 20 ns;
                z <= '1' ; wait for 20 ns;
      end process pz;
py : process is
        begin
                y <= '0' ; wait for 40 ns;
                y <= '1' ; wait for 40 ns;
      end process py;
px : process is
        begin
                x <= '0' ; wait for 80 ns;
                x <= '1' ; wait for 80 ns;
      end process px;
pw : process is
        begin
                w <= '0' ; wait for 160 ns;
                w <= '1' ; wait for 160 ns;
      end process pw;
end architecture behavioural;
--
entity and4 is -- 4 input and gate.
        port( in1, in2, in3, in4 : in bit;
                out1 : out bit
              );
end entity and4;
--
architecture dataflow of and4 is
begin
        out1 <= in1 and in2 and in3 and in4;
end architecture dataflow;
--
entity and3 is -- 3 input and gate.
        port( in1, in2, in3  : in bit;
                out1 : out bit
              );
end entity and3;
--
architecture dataflow of and3 is
begin
        out1 <= in1 and in2 and in3;
end architecture dataflow;
--
```

```
entity or4 is -- 4 input or gate.
        port( in1, in2, in3, in4 : in bit;
                out1 : out bit
              );
end entity or4;
--
architecture dataflow of or4 is
begin
        out1 <= in1 or in2 or in3 or in4;
end architecture dataflow;

entity inv is -- Inverter.
        port( in1 : in bit;
                out1 : out bit
              );
end entity inv;
--
architecture dataflow of inv is
begin
        out1 <= not in1;
end architecture dataflow;
--
entity testbench is  --  Test bench outer entity wrapper.
end entity testbench;
--
library work;
use work.all;
--
architecture structural of testbench is  -- Testbench architecture.
signal b0, b1, b2, b3, zb, yb, xb, wb,a, b, c, d, f : bit;
begin
        d1 : entity test_vector port map(b0, b1, b2, b3);
        d2 : entity inv port map(b0, wb);
        d3 : entity inv port map(b1, xb);
        d4 : entity inv port map(b2, yb);
        d5 : entity inv port map(b3, zb);
        d6 : entity and4 port map(zb, yb, b1, wb, a);
        d7 : entity and4 port map(zb, yb, xb, wb, b);
        d8 : entity and3 port map(zb, yb, b0, c);
        d9 : entity and4 port map(b0, b1, b2, b3, d);
        d10 : entity or4 port map(a, b, c, d, f);
end architecture structural;
```

On entry of this code into the Qucs VHDL text editor the text is colour coded. Unfortunately, the colour coding is lost when printed, or pasted into a word processor,

or a layout package like LaTeX. The structure of the VHDL listing follows the normal convention for text based VHDL simulation. All component entity-architecture models must be defined before they are referenced in other component models. The simulation test bench must be the last entity-architecture model in the VHDL listing. During the VHDL compile phase FreeHDL compiles the component entity-architecture models to the work library[16]. These compiled models are then made available to the simulation test bench through the use of the VHDL ***use*** statement inserted in the listing prior to the testbench architecture statement. Once the VHDL listing for the simulation has been typed into the Qucs VHDL code editor, pressing key F2 starts the simulation process. The simulation duration can be set using the Document Settings in the File dropdown menu (or by pressing the Ctrl+. keys). Any VHDL syntax errors, or indeed typos, are written to file and can be viewed by pressing key F5. Obviously if errors are reported these need to be corrected using the VHDL text editor and the simulation cycle restarted. A typical TimeList output for editor example 1 is shown in Fig. 5.11.

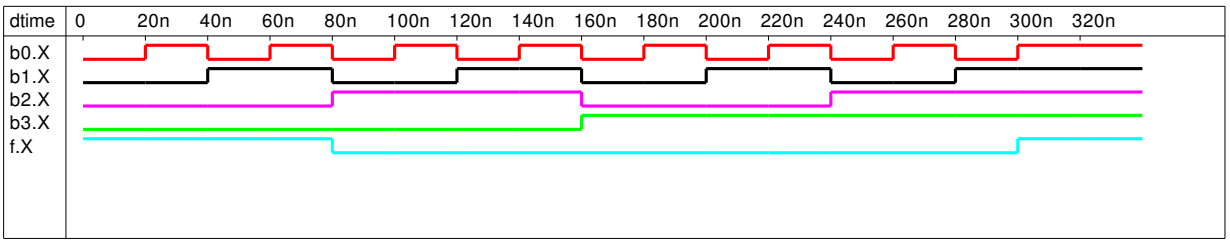


Figure 5.11: Sample simulation waveforms for VHDL editor example 1 design.

- Example 2: VHDL editor example 1 modelled using dataflow VHDL statements.

  The VHDL code for the second example is given in the next listing. The VHDL style chosen to model the circuit is based on VHDL dataflow concurrent signal assignments. The input text vectors are generated using a simple state machine rather than separate process statements. The test vector generator port specification uses entirely single signal bit types and can be easily interfaced, without problems, to other components connected on a Qucs schematic diagram. The procedure for generating schematic capture component symbols from entity - architecture models is introduced in a later section of these notes. The use of bit vector bus constructions is also illustrated in this example. Qucs allows the use of bit vectors as signals or variables in VHDL models provided all signals in the port statement of entity declaration are of type bit only.[17] A typical TimeList output for editor example 2 is shown in Fig. 5.12.

[16] In most VHDL implementations library work is always visible and there is no requirement to make it visible by using the library and use statements. However, FreeHDL appears to need these statements at the linking phase otherwise the VHDL compiler fails.

[17] This is a restriction of Qucs 0.0.9 and will be removed in a later release of the package. Also note signals of type bit vector that are declared in architecture definitions are listed in the TimeList plot signal

```
-- Qucs VHDL editor example 2
--
entity test_vector_a is
        port( RESET, CLOCK : in bit;
              B0, B1, B2, B3 : out  bit
            );
end entity test_vector_a;
--
architecture behavioural of test_vector_a is
signal present_state, next_state : bit_vector(3 downto 0):="1111";
begin
--
p1 : process(CLOCK ) is
        begin
          if  (CLOCK'event and CLOCK='1') then
             present_state <= next_state;
          end if;
       end process p1;
--
p2 : process(RESET,  present_state) is
        begin
         if (RESET = '1' ) then next_state <= "1111";
         end if;
        case present_state is
         when "0000" =>  next_state <= "0001";
         when "0001"  =>  next_state <= "0010";
         when "0010"  =>  next_state <= "0011";
         when "0011"  =>  next_state <= "0100";
         when "0100"  =>  next_state <= "0101";
         when "0101"  =>  next_state <= "0110";
         when "0110"  =>  next_state <= "0111";
         when "0111"  =>  next_state <= "1000";
         when "1000"  =>  next_state <= "1001";
         when "1001"  =>  next_state <= "1010";
         when "1010"  =>  next_state <= "1011";
         when "1011"  =>  next_state <= "1100";
         when "1100"  =>  next_state <= "1101";
         when "1101"  =>  next_state <= "1110";
         when "1110"  =>  next_state <= "1111";
         when "1111"  =>  next_state <= "0000";
        end case;
          B3 <= next_state(3); B2 <= next_state(2);
          B1 <= next_state(1); B0 <= next_state(0);
```

dialogue. However, a text message saying no data results if an attempt is made to display them. Again this limitation will be removed in a later release of Qucs.

```
     end process p2;
end architecture behavioural;
--
library work;
use work.all;
--
entity testbench is
end entity testbench;
--
architecture dataflow of testbench is
signal  reset, clk, b0, b1, b2, b3, zb : bit;
signal  yb, xb, wb,a, b, c, d, f : bit;
begin
p1 : process is
        begin
         clk <= '0'; wait for 10 ns;
         clk <= '1'; wait for 10 ns;
      end process p1;
--
p2 : process is
        begin
          reset <= '1'; wait for 10 ns;
          reset <= '0'; wait for 2000 ns;
     end process p2;
--
d1 : entity test_vector_a  port map(reset, clk, b0, b1, b2, b3);
--
--    Data flow model of combinational circuit
      wb <= not b0;  xb  <= not b1;  yb  <= not b2; zb <= not b3;
       a <= (wb and b1) and (yb and zb);
       b <= (wb and xb) and (yb and zb);
       c <= b0 and (yb and zb);
       d <= (b0 and b1) and (b2 and b3);
       f  <= a or b or c or d;
end architecture dataflow;
```

- Example 3: VHDL editor example 1 modelled using VHDL process statements and variables.

  The VHDL code for the third example is given in the listing at the end of this paragraph. In this example the use of VHDL variables is illustrated. The VHDL code for the vector generator is a little unusual in that rather than using the traditional two process design employing signals, a single process statement employing variables undertakes both the calculation of the next state data and the transfer of the next state information to the present state. This approach is necessary because FreeHDL does not allowed shared variables. Once again in this example only single bit data

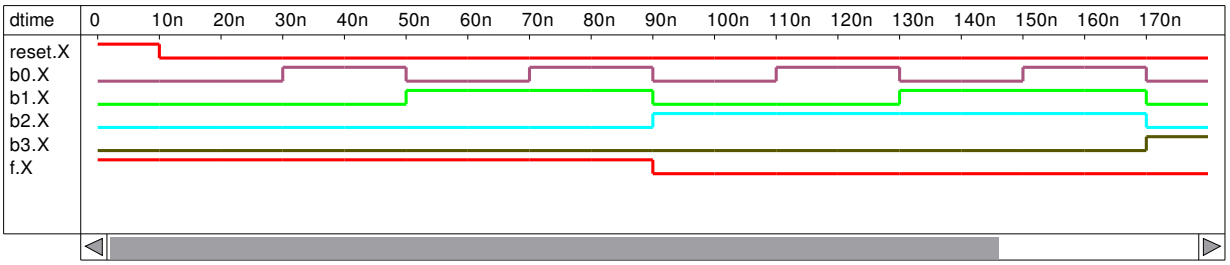


Figure 5.12: Sample simulation waveforms for VHDL editor example 2 design.

is passed via the entity statement to the device under test. The device under test is represented by a truth table encoded in a process statement. This is not the most elegant code but it does serve the purpose of demonstrating the use of different VHDL constructions and data types in Qucs digital simulation. A typical TimeList plot for VHDL editor example 3 is shown in Fig. 5.13. Comparison of the three output plots for the VHDL editor examples indicates that all the simulation results are very similar with some slight differences in the start up phase following the RESET pulse changing from logic '1' to logic '0'. This is probably an effect due to the different initialisation sequences for each of the test vector models.

```
-- Qucs VHDL editor example 3
--
entity test_vector_b is
        port( RESET, CLOCK : in bit;
              B0, B1, B2, B3 : out  bit
            );
end entity test_vector_b;
--
architecture behavioural of test_vector_b is
begin
p1 : process(RESET,  CLOCK) is
     variable present_state, next_state :
                bit_vector(3 downto 0):="0000";
          begin
            if (RESET = '1' ) then next_state := "0000";
            elsif (CLOCK'event and CLOCK='1') then
              present_state := next_state;
            case present_state is
             when "0000"  =>  next_state := "0001";
             when "0001"  =>  next_state := "0010";
             when "0010"  =>  next_state := "0011";
             when "0011"  =>  next_state := "0100";
             when "0100"  =>  next_state := "0101";
             when "0101"  =>  next_state := "0110";
             when "0110"  =>  next_state := "0111";
```

```
                when "0111"  =>  next_state := "1000";
                when "1000"  =>  next_state := "1001";
                when "1001"  =>  next_state := "1010";
                when "1010"  =>  next_state := "1011";
                when "1011"  =>  next_state := "1100";
                when "1100"  =>  next_state := "1101";
                when "1101"  =>  next_state := "1110";
                when "1110"  =>  next_state := "1111";
                when "1111"  =>  next_state := "0000";
              end case;
              end if;
             B3 <= next_state(3); B2 <= next_state(2);
             B1 <= next_state(1); B0 <= next_state(0);
          end process p1;
end architecture behavioural;
--
library work;
use work.all;
--
entity testbench is
end entity testbench;
--
architecture dataflow of testbench is
signal  reset, clk, b0, b1, b2, b3,  f : bit;
begin
p1 : process is
          begin
             clk <= '0'; wait for 10 ns;
             clk <= '1'; wait for 10 ns;
           end process p1;
--
p2 : process is
         begin
           reset <= '1'; wait for 10 ns;
           reset <= '0'; wait for 2000 ns;
        end process p2;
--
d1 : entity test_vector_b  port map(reset, clk, b0, b1, b2, b3);
--
--  Behavioural model of combinational circuit
p3: process(b3, b2, b1, b0) is
        variable SEL : bit_vector (3 downto 0);
           begin
             SEL := b3&b2&b1&b0 ;
             if (SEL = "0010") then f <= '1';
```

```
            elsif (SEL = "0000") then f <= '1';
            elsif (SEL = "1111") then f <= '1';
            elsif (SEL = "0001") then f <= '1';
            elsif (SEL = "0011") then f <= '1';
            else f <= '0';
            end if;
        end process p3;
end architecture dataflow;
```

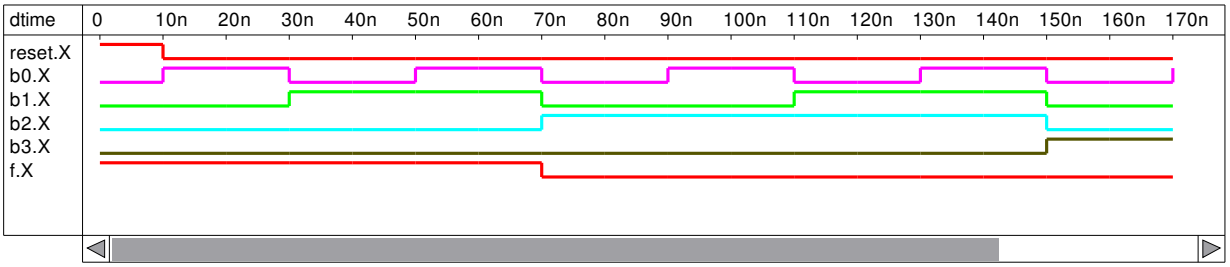


Figure 5.13: Sample simulation waveforms for VHDL editor example 3 design.

5.9.5 Linking VHDL entity-architecture models to Qucs schematic device symbols

VHDL was originally developed as a hardware description language for specifying digital systems. Indeed many engineers still prefer to describe digital systems entirely in VHDL statements rather than use schematic drawings. Once written VHDL code is saved as a text file and becomes the input data for a VHDL compiler/simulation package. Through popular demand a number of digital synthesis/simulator CAD tools[18] have started to include a facility that links VHDL model code to a schematic capture symbol. It is then, of course, possible to use a schematic diagram as the main entry media[19] when designing and simulating a digital design. Qucs release 0.0.9 has such a facility, allowing VHDL code models to be linked to schematic symbols. When drawing digital design schematics, these user defined symbols may be mixed with the Qucs predefined digital symbols and other user defined subcircuit symbols. The process for linking VHDL code to Qucs schematic drawing symbols is straightforward and will be illustrated in these notes through two examples.

- Example 4: A 4 bit test vector pattern generator.

  Shown in Table 5.4 is the VHDL entity-architecture model listing for a 4 bit binary pattern generator. The VHDL code is identical to the test vector code introduced

[18] See for example the XILINX, WebPACK software at `http//www.xilinx.com/ise/logic_design_prod/webpack.htm`.

[19] Please note that at the start of the VHDL simulation process schematic drawings are converted into a VHDL text file.

in the third VHDL editor example. After entering the VHDL entity-architecture model code using the Qucs VHDL editor the finished text is saved in a file with a suitable name and file extension vhdl. Qucs then lists the model under the VHDL project category. Simply clicking on a model name in the VHDL category, with the left hand mouse button, then moving the mouse pointer to a suitable position on a schematic, causes Qucs to move a symbol that represents the model onto the schematic drawing sheet. Placement of the symbol at the position located by the mouse pointer is achieved by clicking the left hand mouse button. The procedure is identical to that used to select and place the Qucs predefined symbols on a schematic drawing. Qucs automatically generates a rectangular symbol with a name called VHDL that has the same number of pins as the port statement listed in the VHDL model entity statement. Each of the pins is given a name that corresponds to a name in the entity statement. Qucs fixes the order of the pins on the generated symbol. It appears that it is not possible to edit this symbol. However, subcircuit in, out or inout port symbols can be attached to symbol VHDL and a user edited symbol generated. Fig. 5.14 shows the Qucs generated VHDL symbol with attached ports for the model listed in Table 5.4. The edited symbol for the 4 bit binary pattern generator is illustrated in Fig. 5.15. Notice that in Fig. 5.15 the order of the pins has been changed to reflect the natural order for a device with it's input pins on the left and output pins on the right. VHDL model symbols can also be generated by placing the VHDL file component, this is located in the digital components viewlist, on a schematic. On editing the VHDL file name property of this device to the name of a VHDL entity-architecture model file, Qucs automatically generates a VHDL symbol. Defining your own symbol then proceeds in a similar fashion to the way described above.

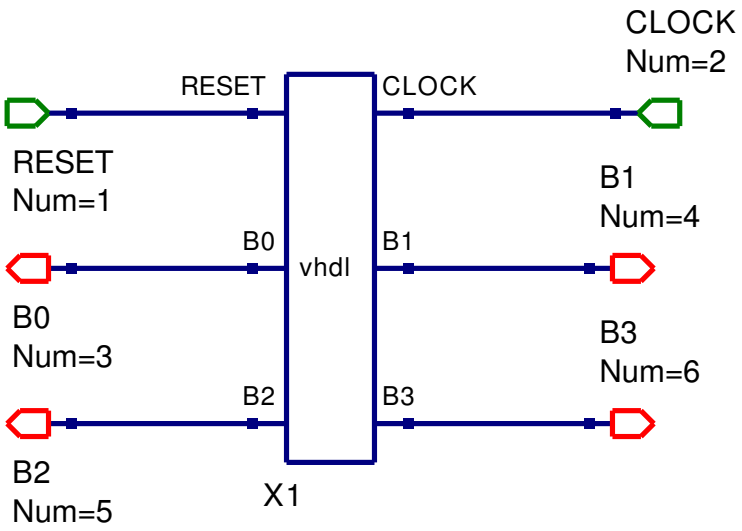


Figure 5.14: Qucs generated VHDL symbol with subcircuit ports for test pattern generator.

```
entity patgen_4bit is
        port( RESET, CLOCK : in bit;
                B0, B1, B2, B3 : out  bit
              );
end entity patgen_4bit;
--
architecture behavioural of patgen_4bit is
begin
p1 : process(RESET,  CLOCK) is
     variable present_state, next_state :
               bit_vector(3 downto 0):="0000";
     begin
      if (RESET = '1' ) then next_state := "0000";
      elsif (CLOCK'event and CLOCK='1') then
      present_state := next_state;
      case present_state is
       when "0000"  =>  next_state := "0001";
       when "0001"  =>  next_state := "0010";
       when "0010"  =>  next_state := "0011";
       when "0011"  =>  next_state := "0100";
       when "0100"  =>  next_state := "0101";
       when "0101"  =>  next_state := "0110";
       when "0110"  =>  next_state := "0111";
       when "0111"  =>  next_state := "1000";
       when "1000"  =>  next_state := "1001";
       when "1001"  =>  next_state := "1010";
       when "1010"  =>  next_state := "1011";
       when "1011"  =>  next_state := "1100";
       when "1100"  =>  next_state := "1101";
       when "1101"  =>  next_state := "1110";
       when "1110"  =>  next_state := "1111";
       when "1111"  =>  next_state := "0000";
      end case;
      end if;
      B3 <= next_state(3); B2 <= next_state(2);
      B1 <= next_state(1); B0 <= next_state(0);
     end process p1;
end architecture behavioural;
```

Table 5.4: VHDL code for a 4 bit pattern generator.

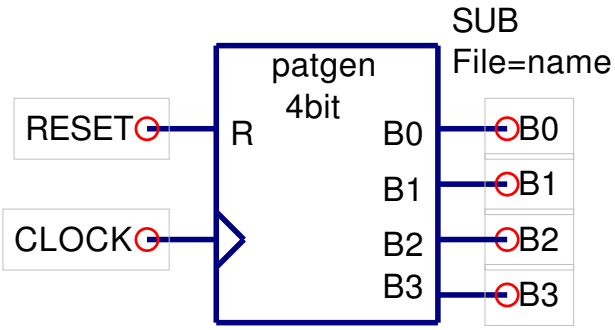


Figure 5.15: User defined 4 bit pattern generator symbol.

```
--  Full adder - 1 bit
entity fulladder is
        port (a, b, cin : in bit;
                sum, cout : out bit
              );
end entity fulladder;
--
architecture dataflow of fulladder is
begin
        sum <= (a xor b) xor cin;
        cout <= (a and b) or (a and cin) or (b and cin);
end architecture dataflow;
```

Table 5.5: VHDL code for a 1 bit full adder.

- Example 5: A 4 bit full adder.

  VHDL model symbols may be combined with either the Qucs predefined digital component symbols or other subcircuit symbols. In this example a VHDL model for a simple one bit full adder is connected four times in a serial fashion to form a 4 bit full adder. The VHDL model code for a simple one bit full adder is given in Table 5.5. The associated symbol diagrams for the one bit full adder are illustrated in Fig. 5.16 and Fig. 5.17.

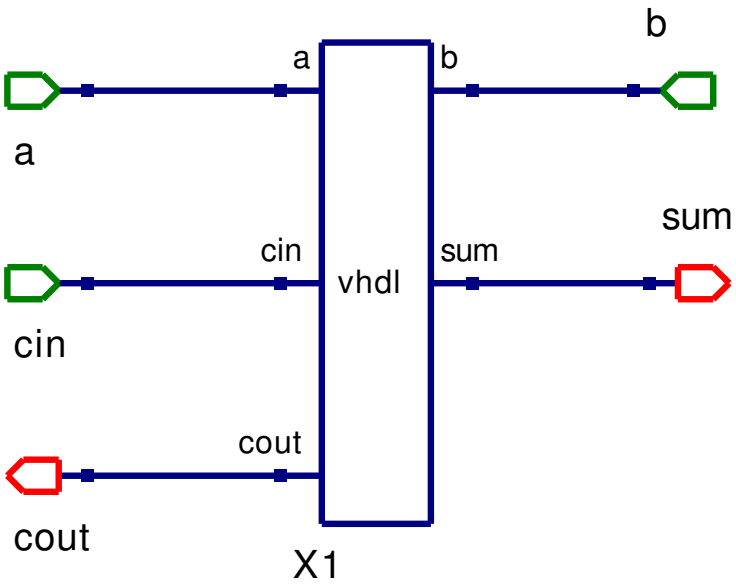


Figure 5.16: Qucs generated VHDL symbol with subcircuit ports for one bit full adder.

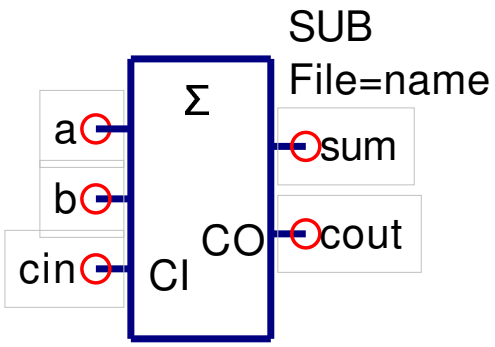


Figure 5.17: User defined one bit full symbol.

Figure 5.18 shows the schematic for a simple 4 bit ripple adder. The corresponding user defined symbol for the 4 bit full adder is given in Fig. 5.19.

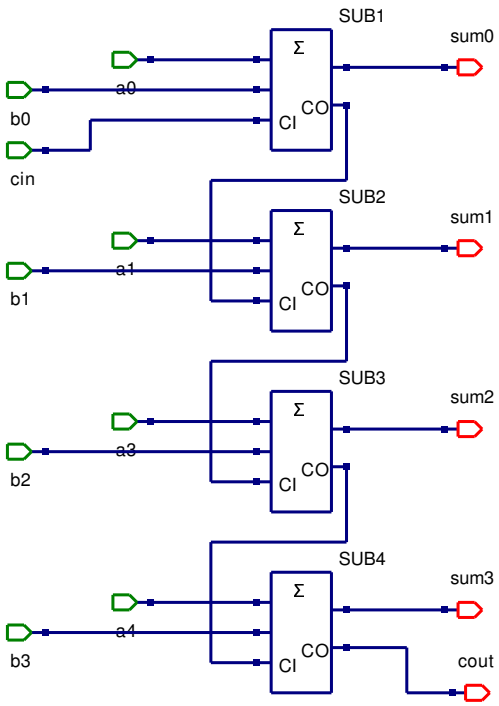


Figure 5.18: 4 bit full adder schematic.

5.9.6 Generating VHDL code from Qucs schematic drawings

Pressing key F2 causes Qucs to simulate the design entered by the Qucs user. The input data for a simulation is either a VHDL text file, saved from the VHDL text editor, or a VHDL code file generated by Qucs using the information encoded on a schematic drawing. In this section of these tutorial notes a larger design is introduced and the resulting VHDL code and simulation results are discussed. The example chosen for this purpose is a 4 bit by 4 bit combinational digital multiplier. Both the 4 bit pattern generator and the 4 bit full adder outlined in the last section form part of the central core of the 4 bit multiplier design and it's associated testbench. Table 5.6 shows the multiplication product table for a 4 bit by 4 bit combinational binary multiplier. Inputs to the device are binary bits a3 a2 a1 a0 and b3 b2 b1 b0. The 4 by 4 multiplier device requires 16 and gates (to generate the multiplier product terms), three four bit full adders (to sum the output r terms) and two 4 bit pattern generators to test the 256 possible input states. The multiplier output is represented in Table 5.6 by r7 r6 r5 r4 r3 r2 r1 and r0. The circuit schematic for the 4 bit by 4 bit multiplier and test bench are given in Fig. 5.20.

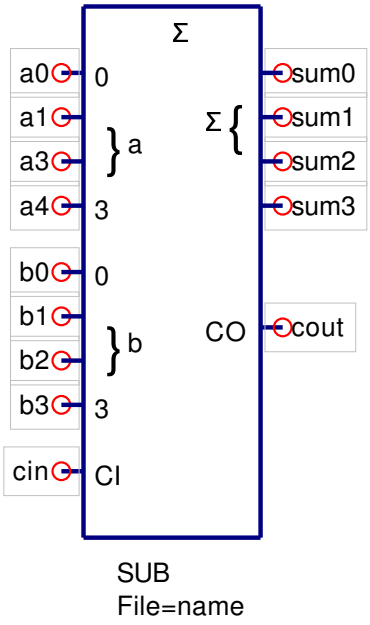


Figure 5.19: User defined 4 bit full adder symbol.

				b3	b2	b1	b0
				a3	a2	a1	a0
				a0b3	a0b2	a0b1	a0b0
			a1b3	a1b2	a1b1	a1b0	
		a2b3	a2b2	a2b1	a2b0		
	a3b3	a3b2	a3b1	a3b0			
r7	r6	r5	r4	r3	r2	r1	r0

Table 5.6: Product table for a 4 bit by 4 bit combinational multiplier.

The VHDL code for this example is presented in the following listing. This listing was generated by Qucs[20]. A small section of the TimeList waveform plot for the digital multiplier is shown in Fig. 5.21. At 1.74 micro seconds input a is "0101", input b is "0111" and the output r is "00100011" which is 35 in decimal. Taking a few random checks of the simulation results indicates that the 4 bit by 4 bit multiplier design works correctly. Notice that the VHDL code generated by Qucs for the 4 bit multiplier does not contain any propagation delay timing data. This could be added to the and gates, if required. However, at this stage in the development of Qucs digital simulation passing timing data, and other parameters, from device symbols generated from VHDL models has not been implemented yet. The use of VHDL generics is an obvious way this could be done. Generics are allowed,

[20]Some readers will have noticed that the naming scheme for internal signal nets is different in the multiplier VHDL listing when compared to the VHDL listings in the first version of these notes. Towards the end of the 0.0.9 development phase the naming convention employed by Qucs was changed to give a more flexible structure.

of course, in text based VHDL simulations.

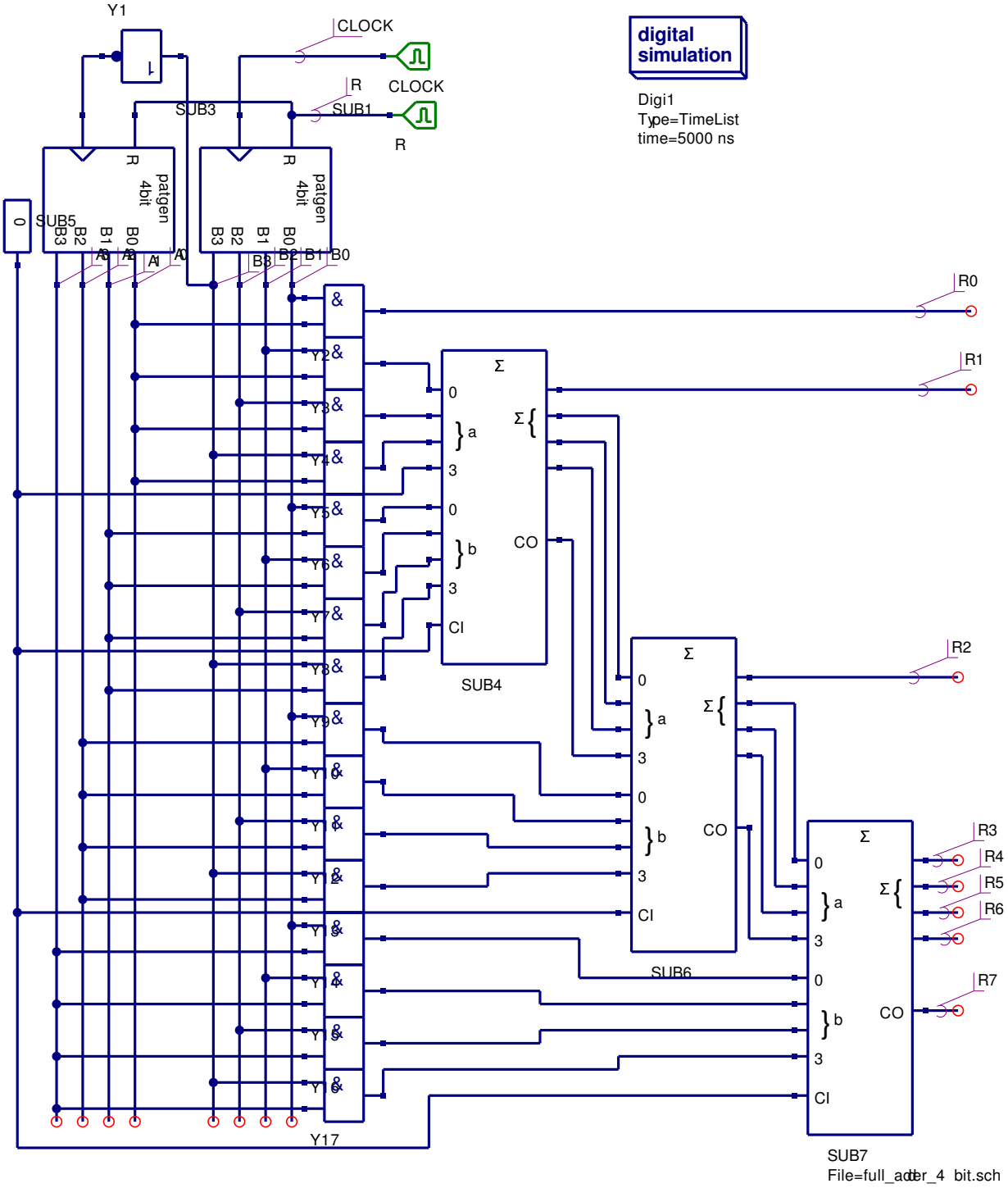


Figure 5.20: A 4 bit by 4 bit combinational digital multiplier.

```
-- Qucs 0.0.9
--  /mnt/hda2/vhdl_comp_lib_prj/multiplier_4bx4bit.sch

entity patgen_4bit is
        port( RESET, CLOCK : in bit;
                B0, B1, B2, B3 : out  bit
              );
end entity patgen_4bit;
--
architecture behavioural of patgen_4bit is
begin
 p1 : process(RESET,  CLOCK) is
        variable present_state, next_state :
             bit_vector(3 downto 0) := "0000";
          begin
            if (RESET = '1' ) then next_state := "0000";
            elsif (CLOCK'event and CLOCK='1') then
              present_state := next_state;
              case present_state is
            when "0000" =>  next_state := "0001";
            when "0001"  =>  next_state := "0010";
            when "0010"  =>  next_state := "0011";
            when "0011"  =>  next_state := "0100";
            when "0100"  =>  next_state := "0101";
            when "0101"  =>  next_state := "0110";
            when "0110"  =>  next_state := "0111";
            when "0111"  =>  next_state := "1000";
            when "1000"  =>  next_state := "1001";
            when "1001"  =>  next_state := "1010";
            when "1010"  =>  next_state := "1011";
            when "1011"  =>  next_state := "1100";
            when "1100"  =>  next_state := "1101";
            when "1101"  =>  next_state := "1110";
            when "1110"  =>  next_state := "1111";
            when "1111"  =>  next_state := "0000";
              end case;
             end if;
           B3 <= next_state(3); B2 <= next_state(2);
           B1 <= next_state(1); B0 <= next_state(0);
        end process p1;
end architecture behavioural;

entity Sub_patgen_4bit is
  port (net_net0: in bit;
        net_net5: in bit;
```

```
        net_outnet_net1: out bit;
        net_outnet_net3: out bit;
        net_outnet_net2: out bit;
        net_outnet_net4: out bit);
end entity;
use work.all;
architecture Arch_Sub_patgen_4bit of Sub_patgen_4bit is
  signal net_net1,
         net_net2,
         net_net3,
         net_net4 : bit;
begin
  net_outnet_net1 <= net_net1 or '0';
  net_outnet_net2 <= net_net2 or '0';
  net_outnet_net3 <= net_net3 or '0';
  net_outnet_net4 <= net_net4 or '0';
  X1: entity patgen_4bit port map (net_net0, net_net5,
        net_net1, net_net3, net_net2, net_net4);
end architecture;


-- logic_zero.vhdl
entity logic_zero is
        port( Y : out bit
              );
end entity logic_zero;
--
architecture dataflow of logic_zero is
begin
        Y <= '0';
end architecture  dataflow;


entity Sub_logic_zero is
  port (net_outnetY: out bit);
end entity;
use work.all;
architecture Arch_Sub_logic_zero of Sub_logic_zero is
  signal netY : bit;
begin
  X1: entity logic_zero port map (netY);
  net_outnetY <= netY or '0';
end architecture;
```

```
--  Full adder - 1 bit
entity fulladder is
        port (a, b, cin : in bit;
                sum, cout : out bit
              );
end entity fulladder;
--
architecture dataflow of fulladder is
begin
        sum <= (a xor b) xor cin;
        cout <= (a and b) or (a and cin) or (b and cin);
end architecture dataflow;


entity Sub_full_adder_1bit is
  port (net_net0: in bit;
        net_net1: in bit;
        net_net2: in bit;
        net_outnet_net3: out bit;
        net_outnet_net4: out bit);
end entity;
use work.all;
architecture Arch_Sub_full_adder_1bit of Sub_full_adder_1bit is
  signal net_net3,
         net_net4 : bit;
begin
  X1: entity fulladder port map (net_net0, net_net1,
           net_net2, net_net3, net_net4);
  net_outnet_net3 <= net_net3 or '0';
  net_outnet_net4 <= net_net4 or '0';
end architecture;


entity Sub_full_adder_4bit is
  port (net_net0: in bit;
        net_net1: in bit;
        net_net2: in bit;
        net_net3: in bit;
        net_net4: in bit;
        net_net5: in bit;
        net_net6: in bit;
        net_net13: in bit;
        net_net7: in bit;
        net_outnet_net8: out bit;
        net_outnet_net9: out bit;
```

```
        net_outnet_net10: out bit;
        net_outnet_net11: out bit;
        net_outnet_net12: out bit);
end entity;
use work.all;
architecture Arch_Sub_full_adder_4bit of Sub_full_adder_4bit is
  signal net_net14,
         net_net15,
         net_net16,
         net_net8,
         net_net9,
         net_net10,
         net_net11,
         net_net12 : bit;
begin
  net_outnet_net8 <= net_net8 or '0';
  net_outnet_net9 <= net_net9 or '0';
  net_outnet_net10 <= net_net10 or '0';
  net_outnet_net11 <= net_net11 or '0';
  net_outnet_net12 <= net_net12 or '0';
  SUB4: entity Sub_full_adder_1bit port map (net_net3, net_net13,
                net_net14, net_net11, net_net12);
  SUB3: entity Sub_full_adder_1bit port map (net_net2, net_net6,
                net_net15, net_net10, net_net14);
  SUB2: entity Sub_full_adder_1bit port map (net_net1, net_net5,
                net_net16, net_net9, net_net15);
  SUB1: entity Sub_full_adder_1bit port map (net_net0, net_net4,
                net_net7, net_net8, net_net16);
end architecture;

entity TestBench is
end entity;
use work.all;

architecture Arch_TestBench of TestBench is
  signal netA0, netA1, netA2, netA3, netR,  netB0,
         netB1, netB2, netB3, netR0, netR1, netR2,
         netR3, netR4, netR5, netR6, netR7, netCLOCK,
         net_net0, net_net1, net_net2, net_net3, net_net4,
         net_net5, net_net6, net_net7, net_net8, net_net9,
         net_net10,net_net11, net_net12, net_net13, net_net14,
         net_net15, net_net16, net_net17, net_net18, net_net19,
         net_net20, net_net21, net_net22, net_net23,
         net_net24 : bit;
begin
```

```
SUB3: entity Sub_patgen_4bit port map (netR, net_net0,
            netA0, netA1, netA2, netA3);
SUB1: entity Sub_patgen_4bit port map (netR, netCLOCK,
            netB0, netB1, netB2, netB3);

R:process
begin
  netR <= '1';  wait for 10 ns;
  netR <= '0';  wait for 2000 ns;
end process;


CLOCK:process
begin
  netCLOCK <= '0';  wait for 10 ns;
  netCLOCK <= '1';  wait for 10 ns;
end process;

net_net0 <= not netB3;
netR0 <= netA0 and netB0;
net_net1 <= netA0 and netB1;
net_net2 <= netA0 and netB2;
net_net3 <= netA0 and netB3;
SUB5: entity Sub_logic_zero port map (net_net4);
net_net5 <= netA1 and netB0;
net_net6 <= netA1 and netB1;
net_net7 <= netA1 and netB2;
net_net8 <= netA1 and netB3;
net_net9 <= netA2 and netB0;
net_net10 <= netA2 and netB1;
net_net11 <= netA2 and netB2;
net_net12 <= netA2 and netB3;
SUB4: entity Sub_full_adder_4bit port map (net_net1, net_net2,
             net_net3, net_net4, net_net5, net_net6, net_net7,
             net_net8, net_net4, netR1, net_net13, net_net14,
             net_net15, net_net16);
SUB6: entity Sub_full_adder_4bit port map (net_net13, net_net14,
             net_net15, net_net16, net_net9, net_net10, net_net11,
             net_net12, net_net4, netR2, net_net17, net_net18,
             net_net19, net_net20);
net_net21 <= netA3 and netB0;
net_net22 <= netA3 and netB1;
net_net23 <= netA3 and netB2;
net_net24 <= netA3 and netB3;
SUB7: entity Sub_full_adder_4bit port map (net_net17, net_net18,
```

```
                net_net19, net_net20, net_net21, net_net22,
                net_net23, net_net24, net_net4, netR3, netR4,
                netR5, netR6, netR7);
end architecture;
```

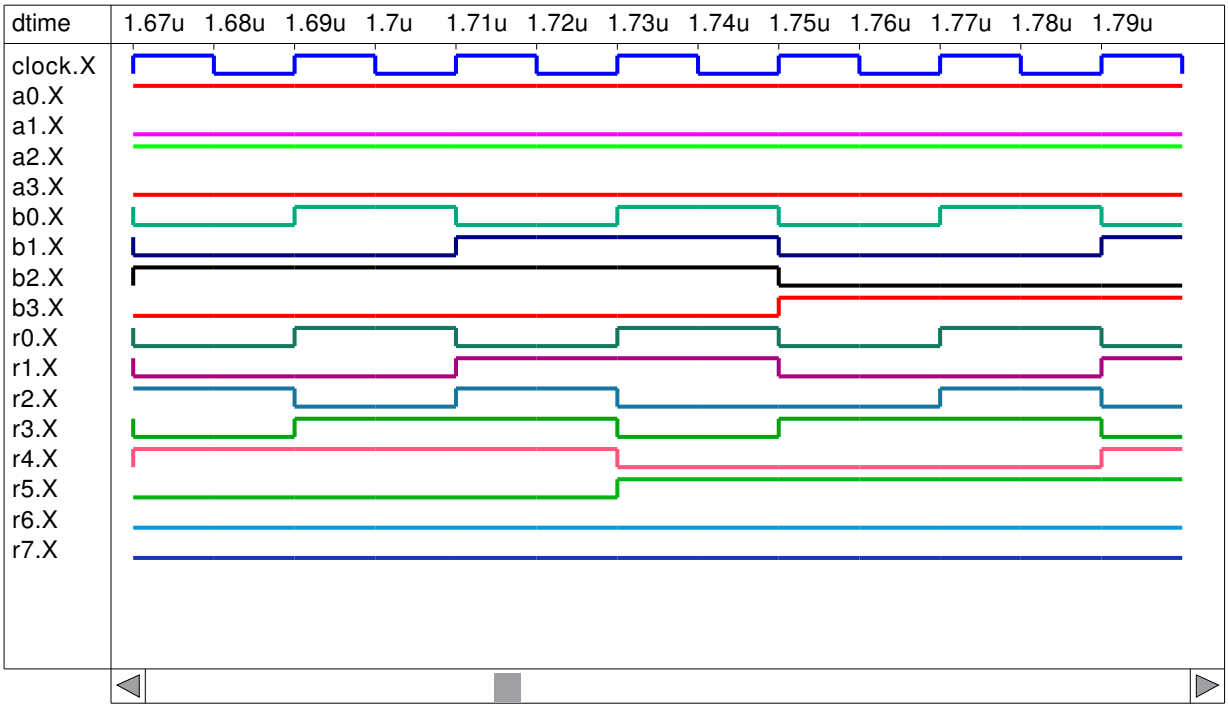


Figure 5.21: A section of the 4 bit by 4 bit combinational digital multiplier TimeList output waveforms.

5.10 Update number two: September 2006

Update number two in this tutorial series reports on the major changes that have taken place to Qucs digital simulation since the first update was posted on the Qucs Web site roughly three months ago. During this period a number of significant, and very critical, extensions have been implemented. Previous releases concentrated on establishing a fundamental base for digital circuit simulation using the VHDL language. The primary vehicle for representing circuit signals being the VHDL bit and bit-vector signal types. The next release of Qucs (version 0.0.10) and FreeHDL (version 0.0.3) extends the allowed signal types to include `IEEE std_logic_1164` nine level logic, integers, and reals. Readers will appreciate that these changes are the result of a great deal of work by the Qucs team and must be considered as very much work in progress because not all the features offered by the FreeHDL implementation of the VHDL language are currently available via the Qucs schematic capture and VHDL text file simulation routes. Although a significant amount

of testing has taken place it is likely that software bugs will come to light as more Qucs users try the new features - if you find a bug please report it by posting a note on the Qucs Web site. Adding new signal types to Qucs digital simulation affects all sections of the simulation route from schematic capture to plotting and tabulating input and output signals. Hence, although it may seem the wrong way round, the place to first implement the necessary changes to accommodate the new signal types is at the simulation results reporting stages of the Qucs package. In release 0.0.10 no attempt has been made to add the new signal types to the schematic capture part of the Qucs package.[21] Recent work on the digital sections of the Qucs package has concentrated on (1) improvements to VHDL language entry using the Qucs colour coded VHDL text editor[22], (2) modifications to FreeHDL which allow a cleaner interface between Qucs and FreeHDL, (3) upgrades to the data conversion of simulation results from the FreeHDL value change dump format to the native Qucs format, and (4) major changes to the results reporting routines that are accessed from the Qucs diagrams icon dialogue. A detailed list of the software changes and bug fixes can be found in the Qucs and FreeHDL change log files.

5.10.1 Simulating VHDL code using Qucs and FreeHDL.

The flow diagram drawn in Fig. 5.10 shows the relationship between Qucs and FreeHDL, and the sequence that takes place during digital circuit simulation. This flow diagram does not however, outline the details of the stages that are performed when converting (1) VHDL circuit code into a machine code simulation program, and (2) simulation output results into a format that can be plotted and tabulated by Qucs. These are illustrated in the flow diagram presented in Fig. 5.22. The shell script qucsdigi controls each of the stages in this sequence. A basic understanding of the process employed by Qucs and FreeHDL is needed if users of the software are to be able to write meaningful VHDL code and simulate it using the two packages. VHDL code is either generated from a schematic diagram automatically by Qucs or entered using the Qucs VHDL text editor. The use of the schematic entry route was described in update one of these tutorial notes. However, a number of readers will probably have spotted that included in the VHDL code generated by Qucs are references to VHDL libraries. The VHDL language uses libraries to provide features that are not specified in the basic language definition but are commonly used by all language processing systems; two such libraries are STD and IEEE. When simulating digital circuits a basic knowledge of the structure of a simulation task and how these employ VHDL libraries is essential. This implies that users of the Qucs/FreeHDL software must appreciate how the system compiles and simulates a VHDL circuit simulation task. Once the VHDL simulation code has been entered via the VHDL text editor clicking the Qucs simulation button runs shell script qucsdigi performing the sequence shown in Fig. 5.22[23].

[21] Adding new signal types to Qucs schematic capture is on the to-do list.

[22] A number of editor bugs have been fixed and it is now possible for users to define their own colour scheme for the various classes of VHDL reserved words and data types.

[23] For the FreeHDL package to operate correctly the directory where the software is installed must be included in the shell PATH from which Qucs is launched.

Program freeehdl-v2cc converts VHDL code into C++ functions. These are then compiled along with a main C++ function. The next stage in the sequence links the compiled object code with the object code from any references to items in the predefined VHDL libraries to produce an executable digital simulation program. This is then run by Qucs outputting a set of simulation results in value change dump (VCD) format[24]. Finally a program called qucsconv converts the VCD simulation results into the Qucs native data format ready for post processing as graphical or tabular diagrams by Qucs.

[24]The value change dump language was originally designed as a simulation waveform interchange format for Verilog HDL. The specification of the VCD format can be found at http://www-ee.eng.hawaii.edu/ msmith/ASICs/HTML/Verilog/LRM/HTML/15/ch15.2.htm

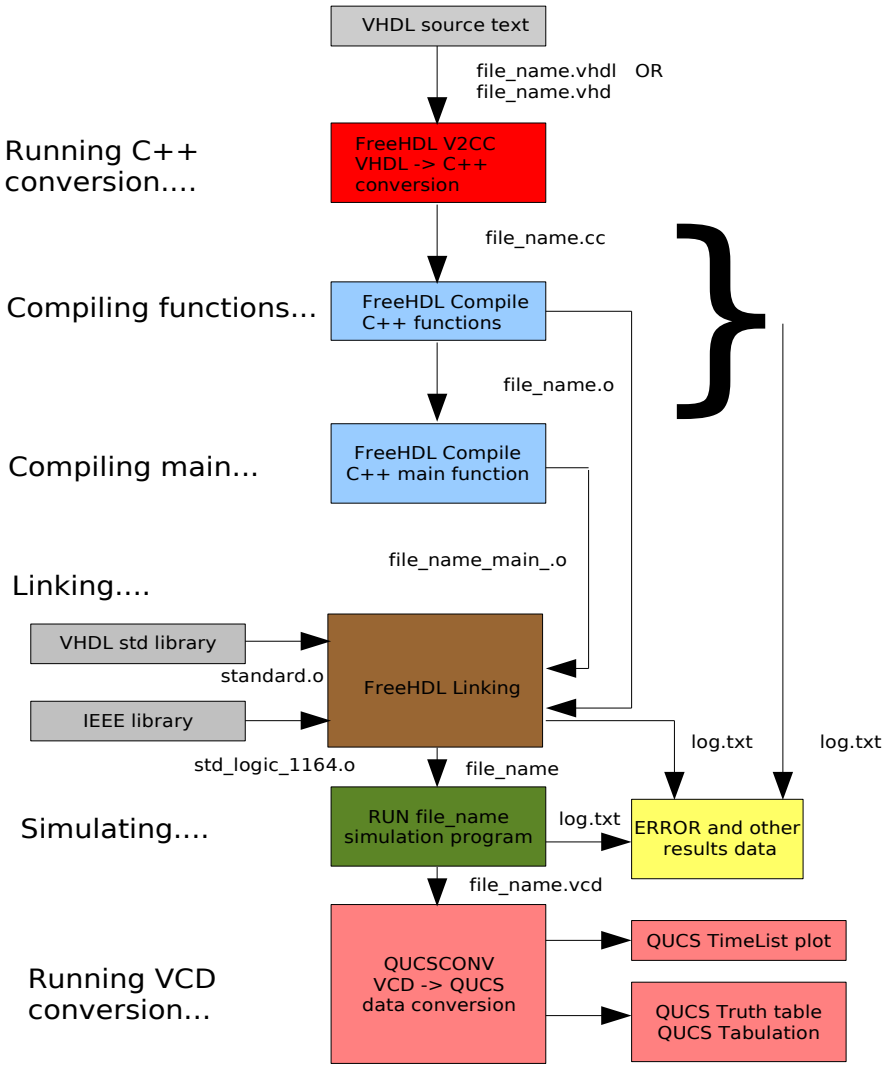


Figure 5.22: Detailed flow diagram showing VHDL code compilation and simulation results processing.

5.10.2 VHDL predefined packages and libraries.

All VHDL language processing systems provide a predefined VHDL package called standard. This package defines many of the fundamental VHDL data types, for example bit, character, integer and real. The predefined types, subtypes and other functions in the package standard are stored in a library called STD. The FreeHDL version of library STD includes an additional VHDL package called textio which is used to input and output signal data from and to files. A second library called IEEE defines (1) multivalued logic signals defined by nine different encoding values, making it possible to model digital circuits that are composed from different technology components, (2) logic signal subtypes and (3) an extensive range of useful functions, procedures and overloaded operators. The FreeHDL version of the IEEE library consists of the following packages:

1. `std_logic_1164`
2. `numeric_bit`
3. `math_real`
4. `numeric_std`
5. `std_logic_arith`
6. `std_logic_unsigned`
7. `vital_timing`

One other library is always defined by VHDL code processing systems namely the work library. This library holds user compiled VHDL entity/architecture design units.

5.10.3 VHDL simulation code structures.

In its most basic form VHDL circuit simulation code is structured as an entity-architecture test bench which includes input signal test information.[25] An example outline of the basic format is

```
entity testbench is
--   entity body statements
end entity testbench;
--
architecture behavioural of testbench is
--   architecture body statements
end architecture behavioural;
```

[25]Test signals are often called test vectors.

VHDL data types, functions and operators in package standard are always visible to VHDL test bench code and reference to their use need not be added explicitly. However, if the test bench entity-architecture uses data types or other items defined in other libraries, for example the `std_logic` type in the IEEE library, then reference to them needs to be added before each entity-architecture pair where they are used. Libraries are referenced using the VHDL *library* and *use* statements. An example showing how these statements are employed is outlined in the following VHDL code segment:

```
library ieee;
use ieee.std_logic_1164.all;
--
entity testbench is
--  entity body statements
end entity testbench;
--
architecture behavioural of testbench is
--  architecture body statements
end architecture behavioural;
```

Here the VHDL code word *all* signifies that all items in a specific library are to be made available for use in the following entity/architecture pair; testbench in the above example. If more than one library is to be used then a library/use statement is needed for each library reference. Most complete VHDL circuit simulation programs consist of more than one entity/architecture pair. In such cases the circuit test bench, with its signal test vectors, must be the last entry in the program. An example of a more complex VHDL program structure is

```
library ieee;
use ieee.std_logic_1164.all;
--
entity comp1 is
--  entity body statements
end entity comp1;
--
architecture behavioural of comp1 is
--  architecture body statements
end architecture behavioural;
--
library ieee;
use ieee.std_logic_1164.all;
--
entity comp2 is
--  entity body statements
end entity comp2;
--
architecture behavioural of comp2 is
--  architecture body statements
```

```
end architecture behavioural;

--
library ieee;
use ieee.std_logic_1164.all;
--
use work.all;
--
entity testbench is
--  entity body statements
end entity testbench;
--
architecture behavioural of testbench is
--  architecture body statements
end architecture behavioural;
```

During the conversion of VHDL code to a machine code simulation program each entity/architecture pair, prior to the final test bench entry, is compiled as a separate design unit and stored in the work library[26]. Compiled design units held in the work library can be referenced in other entity/architecture models provided the VHDL statement *use work.all;*[27] is inserted in the VHDL simulation code prior to each entity/architecture statement where they are referenced.

[26]The testbench entity/architecture pair is also, of course, compiled but this design unit is the one that is run as the executable simulation program.

[27]References to individual items are also allowed by inserting, for example, *use.work.comb1; use.work.comb2;* in the VHDL code.

5.10.4 VHDL data types.

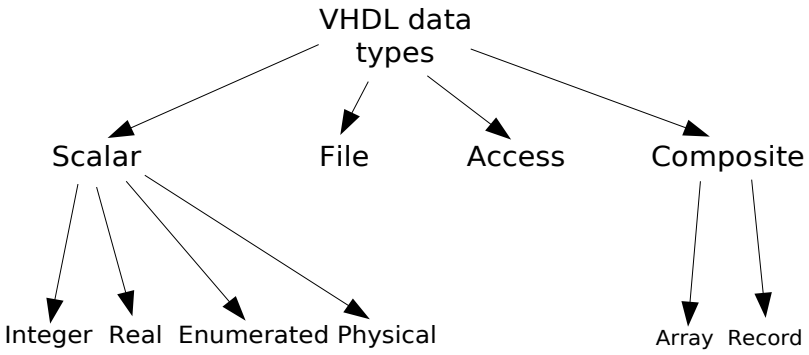


Figure 5.23: VHDL data types

The chart shown in Fig. 5.23 indicates the different data types that are available in the VHDL language. FreeHDL implements all these data types. In practical circuit simulation the different VHDL data types are normally used to specify (1) signals, (2) variables and (3) constants[28]. During simulation Qucs/FreeHDL automatically stores the values of integer, real and enumerated bit signals as simulation time progresses. Furthermore, `bit_vector` and IEEE signal types including `std_logic_vector` are also stored. Signals of these types are then available for plotting and tabulation using the Timing, Truth table, Tabular and Cartesian output diagrams. Selected elements in user defined composite signals, those that are stored in arrays for example[29], can be assigned to the basic signal types then displayed.[30]. An example of how this is done is given in later sections of these update tutorial notes. Note - the values of variables and constants are not recorded during simulation.

[28]Type file is of course different in that it is used to store either test vectors, component data such as ROM contents and output simulation results.

[29]Please note that signal types based on the composite type record will probably cause the Qucs simulation cycle to fail - work on this data type has been added to the to-do list.

[30]Qucs/FreeHDL also automatically collects waveform data for composite signals based on arrays of bit and IEEE signal types. However, in the case of large arrays care is needed when plotting or tabulating these directly because the entire contents of an array is output each time a signal is displayed.

5.10.5 An example VHDL simulation employing integer signals.

The following VHDL code demonstrates how the integer data type can be used to represent signals. In this example signals A, B change state on the rising edge of clock clk. The code tests the addition of integer signals and constants using arithmetic operators defined in library STD.[31] The results from this simulation are shown in Fig. 5.24.

```
-- A very basic test of data type integer.
entity testbench is
end entity testbench;
--
architecture behavioural of testbench is
signal A, B, C :  integer := 0;
signal clk : bit;
begin
p0 : process is  -- Generate clock signal.
        begin
            clk <= '0';    wait for 10 ns;
            clk <= '1';    wait for 10 ns;
        end process p0;
--
p1 : process (clk) is
        begin
          if (clk'event and clk='1') then
            A <= A + 1;
            B <= B + 2;
          end if;
        end process p1;
            C <= A + B ;
end architecture behavioural;
```

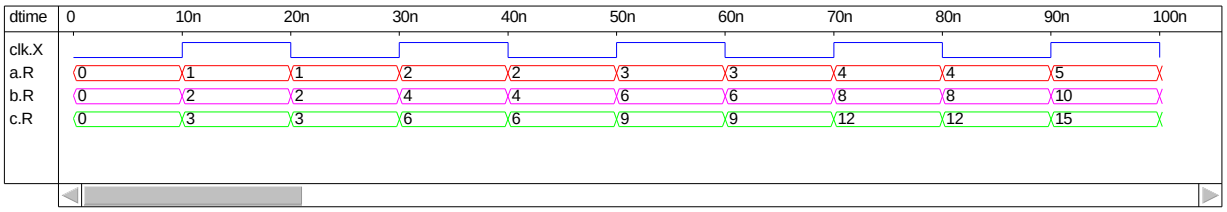


Figure 5.24: Output results for a simple test bench example employing integer signals.

5.10.6 Multivalued logic.

Although signal types bit and bit-vector are widely employed when simulating digital systems one of their great weaknesses is the fact that it is difficult to represent signal bus

[31] The specification for the FreeHDL library STD can be found in text file freehdl-0.0.3/std/standard.vhdl.

systems simply using only logic '0' and logic '1' signal encoding. Moreover, circuits where bus signal contention occurs often result in simulation failure. The IEEE `std_logic_1164` package overcomes this limitation through the introduction of a multivalued logic system which defines nine different logic values to represent signal types and signal strengths. Not only is the bus contention problem solved through logic resolving functions but the multivalued logic system allows devices constructed from different manufacturing technologies to be simulated at the same time, ensuring that the simulation process mirrors real circuit design practices. The next two simulation examples introduce the nine value logic system and demonstrate it's use in the design of digital bus systems. Signals of type real are also introduced to show their representation by Qucs. Listed below is the VHDL code for a basic simulation which generates a set of IEEE `std_logic`, integer and real signals. Figure 5.25 illustrates how the Qucs Timing diagram displays different signal types. A section of tabulated results are also given in Fig. 5.26.

```
library ieee;
use ieee.std_logic_1164.all;
--
entity testbench is
end entity testbench;
--
architecture behavioural of testbench is
signal clk : bit;
signal bv1 : bit_vector(8 downto 0);
signal stdl1 : std_logic_vector(8 downto 0);
signal INT1 : integer := 0;
signal INT2 : integer := 99;
signal R1 : real := 0.33;
signal R2 : real := 99.0;
signal R3 : real := 0.0;
signal R4 : real := 0.0;
begin
p0 : process is
      begin
        clk <= '0'; wait for 10 ns;
        clk <= '1'; wait for 10 ns;
      end process p0;
--
p1 : process(clk) is
        variable  v1 :  integer := 0;
        begin
          if (clk'event and clk = '1') then
                v1 := v1+1;
                case v1 is
                  when 1 => bv1 <= "000000000"; stdl1 <= "000000000";
```

```
                    when 2 => bv1 <= "000000001"; stdl1 <= "000000001";
                    when 3 => bv1 <= "000000011"; stdl1 <= "00000001X";
                    when 4 => bv1 <= "000000111"; stdl1 <= "0000001XZ";
                    when 5 => bv1 <= "000001111"; stdl1 <= "000001XZU";
                    when 6 => bv1 <= "000011111"; stdl1 <= "00001XZUW";
                    when 7 => bv1 <= "000111111"; stdl1 <= "0001XZUWL";
                    when 8 => bv1 <= "001111111"; stdl1 <= "001XZUWLH";
                    when 9 => bv1 <= "111111111"; stdl1 <= "01XZUWLH-";
                    when others => v1 := 0;
                end case;
        end if;
          end process p1;
p3 : process (clk) is
        begin
          if (clk'event and clk='1')  then
              INT1 <= INT1 + 1;
              INT2 <= INT2 -20;
          end if;
--
          if(INT1 >= 9) then
              INT1 <= 0;
              INT2 <= 99;
         end if;
       end process p3;
--
p4 : process (clk) is
         Variable V2 : real;
        begin
          if (clk'event and clk='1')  then
              R1 <= R1 + 1.0;
              R2 <= R2 -20.0;
              R3 <= R1*R2;
              R4 <= R2/(R1+0.0001);
          end if;
--
          if(R1 >= 20.0) then
              R1 <= 0.0;
              R2 <= 99.0;
         end if;
       end process p4;
end architecture behavioural;
```

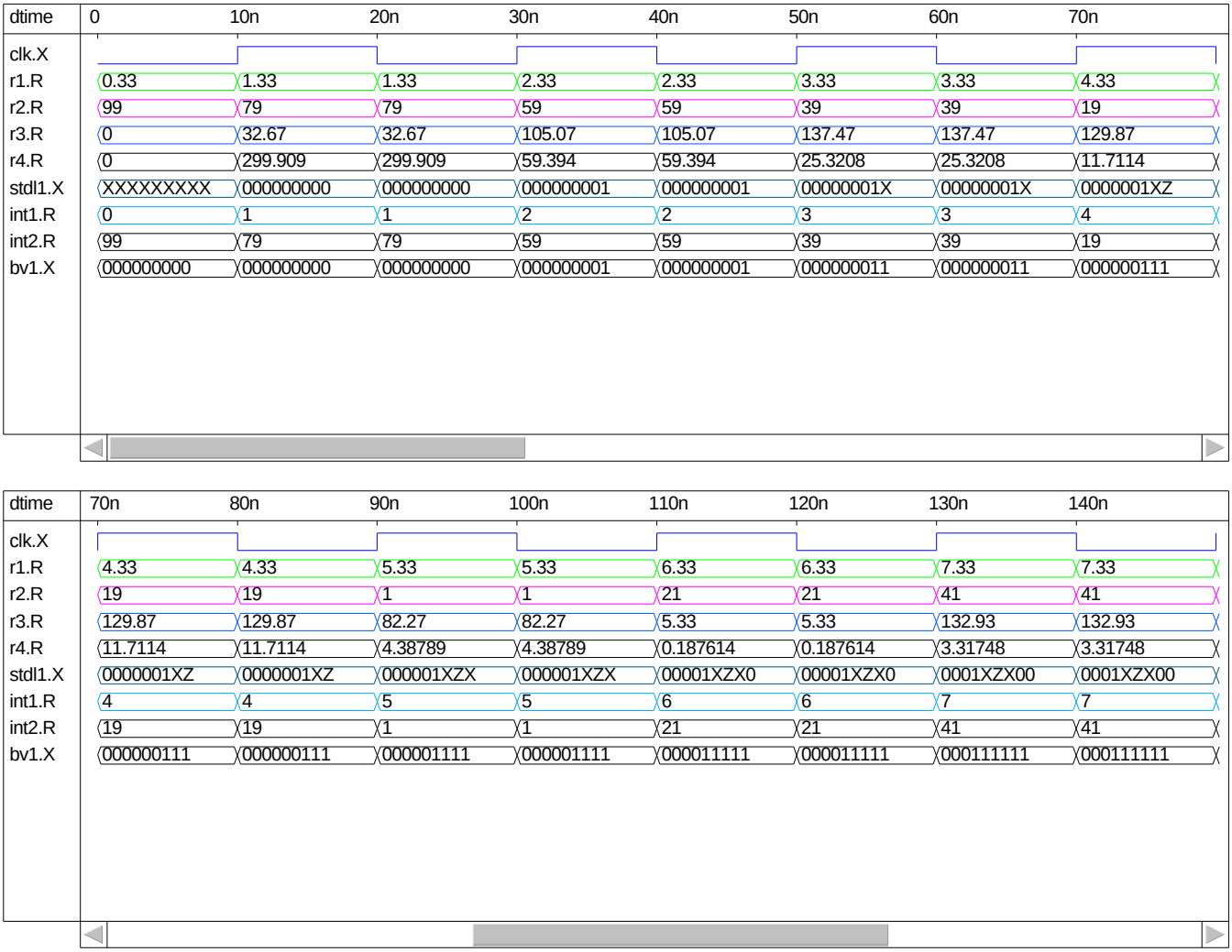


Figure 5.25: Output results illustrating the TimeList representation of signals.

The VCD waveform interchange standard encodes digital signals as four different logic levels. These are '0', '1', 'Z' (high impedance) and 'X' (unknown). Table 5.7 lists how the nine `ieee.std_logic` signal levels are represented using the VCD format. Until the VCD standard is revised the Qucs/FreeHDL package is restricted to displaying simulation output data using the basic '0', '1', 'Z' and 'X' signal encoding. The next example shows how the IEEE `std_logic` signal type can be used to simulate bus logic. The demonstration has been kept simple in order to keep the VHDL code short. The code fragment simulates two tri-state buffers which pass their outputs to bus drivers who's outputs connect on a common signal bus. The bus drivers ensure that the outputs from the tri-state buffers are kept separate before combining onto the common bus line. This allows the output signals from the tri-state buffers and the combined signal to be plotted separately. The resulting waveforms clearly show the `std_logic` resolution function in operation, see Fig. 5.27 . Note

VHDL signal levels	VCD
'0' Forcing logic 0	'0'
'1' Forcing logic 1	'1'
'X' Forcing unknown	'X'
'Z' High impedance	'Z'
'U' Uninitialised	'X'
'W' Weak unknown	'0'
'L' Weak logic 0	'0'
'H" Weak logic 1	'1'
'-' Don't care	'X'

Table 5.7: IEEE multivalue logic and VCD representation.

the effect of the 7 ns delay on the plotted waveforms and the use of the VHDL generic statement to set the invert device delay value.

```
-- Demonstration of a simple bus structure using
-- the IEEE std_logic data type.
library ieee;
use ieee.std_logic_1164.all;
--
entity buf  is
        generic(delay : time := 0 ns);
        port (in1, control  : in std_logic;
              out1 : out std_logic
                    );
end entity buf;
architecture behavioural of buf is
begin
p0 : process (in1, control) is
        begin
          if (control = '1') then out1 <= in1 after delay;
          else out1 <= 'Z';
         end if;
       end process p0;
end architecture behavioural;
--
library ieee;
use ieee.std_logic_1164.all;
--
entity invert is
        generic(delay : time := 0 ns);
        port (in1 : in std_logic;
```

```
              out1 : out std_logic
                   );
end entity invert;
--
architecture behavioural of invert is
begin
   out1 <=  not in1 after delay;
end architecture behavioural;
--
library ieee;
use ieee.std_logic_1164.all;
--

entity buf2 is
  port (in1 : in std_logic;
          out1 : out std_logic
         );
end entity buf2;
--
architecture dataflow of buf2 is
begin
  out1 <= in1;
end architecture dataflow;
--
library ieee;
use ieee.std_logic_1164.all;
--
use work.all;
--
entity testbench is
end entity testbench;
--
architecture structural of testbench is
signal data_in_1, data_in_2 : std_logic;
signal data_out_1,  data_out_2 : std_logic;
signal data_control,  control_buf1 : std_logic;
signal result : std_logic;
--
begin
p0 : process is
       begin
         data_in_1 <= '0'; wait for 5 ns;
         data_in_1 <= '1'; wait for 5 ns;
      end process p0;
--
```

```
      data_in_2 <= not data_in_1;
--
p1 : process is
       begin
         data_control <= '1'; wait for 40 ns;
         data_control <= '0'; wait for 40 ns;
      end process p1;
--
c1g1 : entity buf     port map(in1 => data_in_1, control => data_control,
                               out1 => data_out_1);
c1g2 : entity invert  generic map (delay => 7 ns)
                      port map(in1 => data_control,out1 => control_buf1);
c1g3 : entity buf     port map(in1 => data_in_2, control => control_buf1,
                               out1 => data_out_2);
c1g4 : entity  buf2   port map(in1 => data_out_1, out1 => result);
c1g5 : entity  buf2   port map(in1 => data_out_2, out1 => result);
--
end architecture structural;
```

dtime	clk.X	int1.R	int2.R	r1.R	r2.R	r3.R	r4.R	bv1.X	stdl1.X
0	0	0	99	0.33	99	0	0	000000000	XXXXXXXXX
1e-8	1	1	79	1.33	79	32.7	300	000000000	000000000
2e-8	0	1	79	1.33	79	32.7	300	000000000	000000000
3e-8	1	2	59	2.33	59	105	59.4	000000001	000000001
4e-8	0	2	59	2.33	59	105	59.4	000000001	000000001
5e-8	1	3	39	3.33	39	137	25.3	000000011	00000001X
6e-8	0	3	39	3.33	39	137	25.3	000000011	00000001X
7e-8	1	4	19	4.33	19	130	11.7	000000111	0000001XZ
8e-8	0	4	19	4.33	19	130	11.7	000000111	0000001XZ
9e-8	1	5	-1	5.33	-1	82.3	4.39	000001111	000001XZX
1e-7	0	5	-1	5.33	-1	82.3	4.39	000001111	000001XZX
1.1e-7	1	6	-21	6.33	-21	-5.33	-0.188	000011111	00001XZX0
1.2e-7	0	6	-21	6.33	-21	-5.33	-0.188	000011111	00001XZX0
1.3e-7	1	7	-41	7.33	-41	-133	-3.32	000111111	0001XZX00
1.4e-7	0	7	-41	7.33	-41	-133	-3.32	000111111	0001XZX00
1.5e-7	1	8	-61	8.33	-61	-301	-5.59	001111111	001XZX001
1.6e-7	0	8	-61	8.33	-61	-301	-5.59	001111111	001XZX001
1.7e-7	1	9	-81	9.33	-81	-508	-7.32	111111111	01XZX001X
1.8e-7	0	0	99	9.33	-81	-508	-7.32	111111111	01XZX001X
1.9e-7	1	1	79	10.3	-101	-756	-8.68	111111111	01XZX001X

	clk.X	int1.R	int2.R	r1.R	r2.R	r3.R	r4.R	bv1.X	stdl1.X
00000	0	0	99	0.33	99	0	0	0 0 0 0 0 0 0 0 0	X X X X X X X X X
00001	1	1	79	1.33	79	32.67	299.909	0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0
00010	0	1	79	1.33	79	32.67	299.909	0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0
00011	1	2	59	2.33	59	105.07	59.394	0 0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 0 1
00100	0	2	59	2.33	59	105.07	59.394	0 0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 0 1
00101	1	3	39	3.33	39	137.47	25.3208	0 0 0 0 0 0 0 1 1	0 0 0 0 0 0 0 1 X
00110	0	3	39	3.33	39	137.47	25.3208	0 0 0 0 0 0 0 1 1	0 0 0 0 0 0 0 1 X
00111	1	4	19	4.33	19	129.87	11.7114	0 0 0 0 0 0 1 1 1	0 0 0 0 0 0 1 X Z
01000	0	4	19	4.33	19	129.87	11.7114	0 0 0 0 0 0 1 1 1	0 0 0 0 0 0 1 X Z
01001	1	5	1	5.33	1	82.27	4.38789	0 0 0 0 0 1 1 1 1	0 0 0 0 0 1 X Z X
01010	0	5	1	5.33	1	82.27	4.38789	0 0 0 0 0 1 1 1 1	0 0 0 0 0 1 X Z X
01011	1	6	21	6.33	21	5.33	0.187614	0 0 0 0 1 1 1 1 1	0 0 0 0 1 X Z X 0
01100	0	6	21	6.33	21	5.33	0.187614	0 0 0 0 1 1 1 1 1	0 0 0 0 1 X Z X 0
01101	1	7	41	7.33	41	132.93	3.31748	0 0 0 1 1 1 1 1 1	0 0 0 1 X Z X 0 0
01110	0	7	41	7.33	41	132.93	3.31748	0 0 0 1 1 1 1 1 1	0 0 0 1 X Z X 0 0
01111	1	8	61	8.33	61	300.53	5.59338	0 0 1 1 1 1 1 1 1	0 0 1 X Z X 0 0 1
10000	0	8	61	8.33	61	300.53	5.59338	0 0 1 1 1 1 1 1 1	0 0 1 X Z X 0 0 1
10001	1	9	81	9.33	81	508.13	7.32284	1 1 1 1 1 1 1 1 1	0 1 X Z X 0 0 1 X
10010	0	0	99	9.33	81	508.13	7.32284	1 1 1 1 1 1 1 1 1	0 1 X Z X 0 0 1 X
10011	1	1	79	10.33	101	755.73	8.68158	1 1 1 1 1 1 1 1 1	0 1 X Z X 0 0 1 X
10100	0	1	79	10.33	101	755.73	8.68158	1 1 1 1 1 1 1 1 1	0 1 X Z X 0 0 1 X

Figure 5.26: Output results illustrating tabular representation of signals.

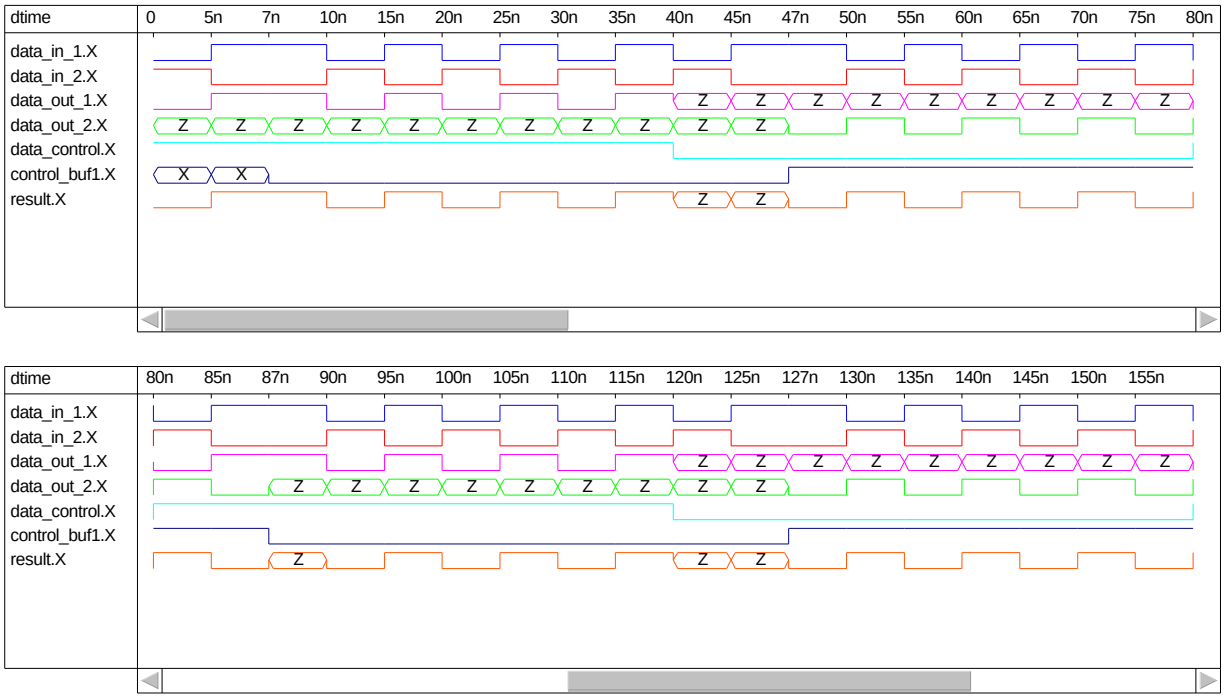


Figure 5.27: Signal waveforms for the simple bus example.

5.10.7 Run debugging of VHDL simulation code.

The VHDL language has a number of built in features that allow the debugging of VHDL code at simulation time. In this section the VHDL reserved words *assert, report* and *severity* are introduced and their use as code debugging aids explained by way of a more detailed design example. In the previous digital tutorial update a structural design of a 4 bit digital multiplier was introduced as an example that employed the Qucs schematic capture digital simulation route. The next example extends the previous multiplier design to 16 bits. However, at a structural level the larger multiplier becomes very detailed and it's design can be prone to error. To demonstrate the power of VHDL the 16 bit multiplier has been redesigned at a functional level. A block diagram of the multiplier simulation test bench is given in Fig. 5.28: firstly a clock strobes a data generator unit which generates a sequence of integer numbers. These are converted to 16 `bit_vectors` and applied to the 16 bit multiplier unit as inputs x and y; secondly the 16-bit multiplier on sensing a change in inputs x or y converts these signals from 16 `bit_vectors` to integers, multiples them and finally converts the integer result to 32 `bit_vector` output `Res_bit`. Although standard library STD defines arithmetic operations for integers it does not provide functions for the conversion of integers to `bit_vectors` or the reverse operation. The following VHDL listing gives the complete simulation test bench program for the 16 bit multiplier including the required data conversion functions. VHDL debug or message reporting code using the reserved words assert, report and severity have been added to the `data_generator`

and `functional_multiplier` architecture code. During simulation these text strings, and the simulation time when they were actioned, are written to the Qucs log.txt file, giving a trace record of the simulation activity. In cases where an error occurs at severity level failure the simulation will terminate. FreeHDL allows VHDL report statements without an accompanying assert statement.[32] A typical Timing diagram plot for this design is shown in Fig. 5.29

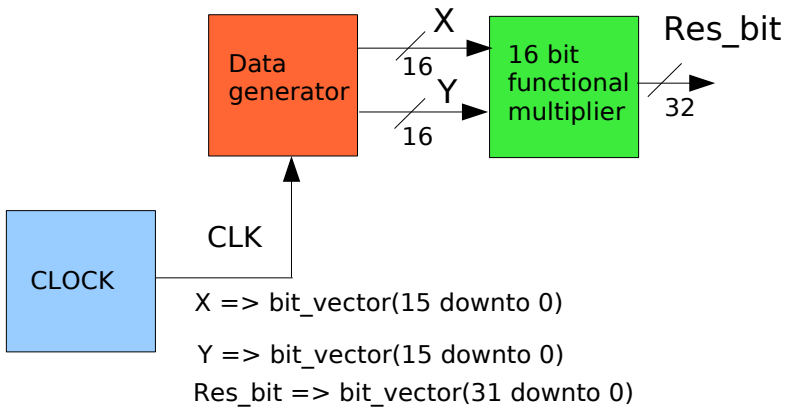


Figure 5.28: Block diagram of a 16 bit functional multiplier.

```
-- 16 bit digital multiplier example.
-- Simulation trace using assert, report and severity statements.
--
entity clock is
    port( clk : out bit);
end entity clock;
--
architecture behavioural of clock is
begin
p0 : process  is
        begin
            clk <= '0' ; wait for 10 ns;
            clk <= '1' ; wait for 10 ns;
```

[32] One of the changes at the 1993 revision of the IEEE VHDL 1076-1987 standard was to allow report statements without the previous mandatory assert clause. FreeHDL attempts to comply with the 1993 revision.

```
       end process p0;
end architecture behavioural;
--
entity data_generator is
     port( clk : in bit;
           x, y : out bit_vector(15 downto 0)
          );
end entity data_generator;
--
architecture behavioural of data_generator is
type mem_array_16  is array(1 to 8) of integer;
signal count : integer := 0;
--
function integer_to_vector_16(int_no : integer) return bit_vector  is
variable ni : integer;
variable return_value : bit_vector(15 downto 0);
begin
   assert (ni < 0)
      report "Function integer_to_vector_32: integer number must be >= 0"
      severity failure;
   ni := int_no;
   for i in return_value'Reverse_Range loop
      if ( (ni mod 2 ) =1 ) then return_value(i) := '1';
      else return_value(i) := '0';
      end if ;
      ni := ni/2;
   end loop;
   return return_value;
end integer_to_vector_16;
--
begin
p1 : process(clk) is
        variable xi : mem_array_16 := (1, 2, 3, 4, 5, 6, 7, 8);
        variable yi : mem_array_16 := (2, 4, 6, 8, 10, 12, 14, 16);
        variable xh, yh : integer;
        variable counti : integer;
      begin

        counti := count+1;
            if (counti > 8 ) then
                counti := 1;
            end if;
               xh := xi(counti);
               yh := yi(counti);
               x <= integer_to_vector_16(xh);
```

```
                y <= integer_to_vector_16(yh);
                count <= counti;
                report "In process p1.data_generator.";
      end process p1;
end architecture behavioural;
--
--
entity functional_multiplier is
     port(  x, y : in bit_vector(15 downto 0);
             res_bit : out bit_vector(31 downto 0)
            );
end entity functional_multiplier;
--
--
architecture behavioural of functional_multiplier is
--
function vector_to_integer(v1 : bit_vector)  return integer is
variable return_value : integer :=0;
alias v2 : bit_vector(v1'length-1 downto 0) is v1;
begin
   for i in v2'high downto 1 loop
      if (v2(i) = '1') then
                return_value := (return_value+1)*2;
      else
                return_value := return_value*2;
     end if;
   end loop;
   if v2(0) = '1' then return_value:= return_value+1;
   end if;
return return_value;
end vector_to_integer;
--
function integer_to_vector_32(int_no : integer) return bit_vector  is
variable ni : integer;
variable value : bit_vector(31 downto 0);
begin
   assert (ni < 0)
      report "Function integer_to_vector_32: integer number must be >= 0"
      severity failure;
   ni := int_no;
   for i in 0 to 31 loop
      if ( (ni mod 2 ) =1 ) then value(i) := '1';
      else value(i) := '0';
      end if ;
      if ni > 0 then ni := ni/2;
```

```
      else ni := (ni-1)/2;
      end if;
   end loop;
   return value;
end integer_to_vector_32;
--
begin
p0 : process (x,y) is
        variable xi, yi, prod_mult : integer;
        begin
         xi := vector_to_integer(x);
         yi := vector_to_integer(y);
         prod_mult := xi*yi;
         res_bit <= integer_to_vector_32(prod_mult);
            report "In process p1.functional_multiplier";
       end process p0;
end architecture behavioural;
--
entity test2_vhdl_1 is
end entity test2_vhdl_1;
--
architecture behavioural of test2_vhdl_1 is
signal clk : bit;
signal x, y : bit_vector(15 downto 0);
signal res_bit : bit_vector(31 downto 0);
--
begin
d1 : entity work.clock port map (clk);
d2 : entity work.data_generator port map(clk, x, y);
d3 : entity work.functional_multiplier port map ( x, y, res_bit);

end architecture behavioural;
```

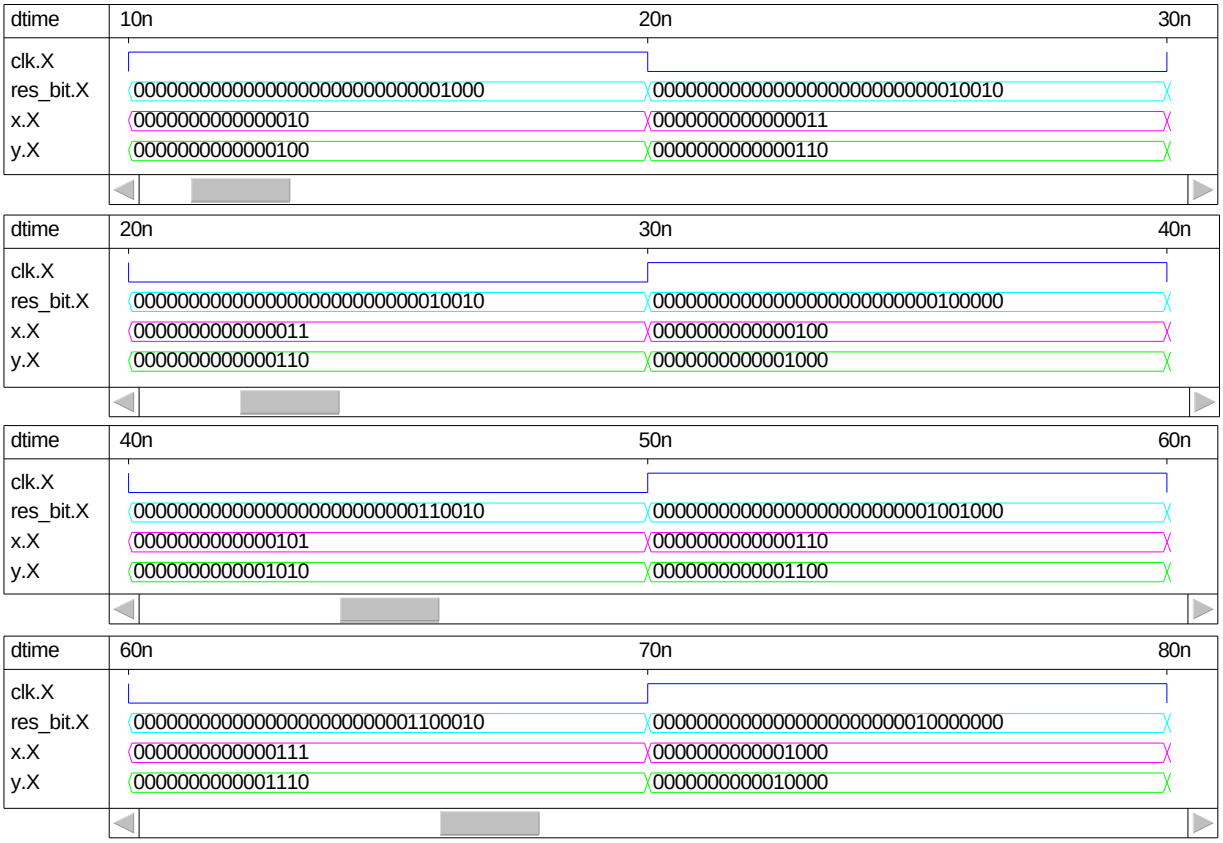


Figure 5.29: Typical timing diagram for the 16 bit functional multiplier.

More advanced output debug messages, and results tables, can be written to Qucs message file log.txt by using the predefined data handling routines in STD library package textio[33]. This package contains functions for reading and writing STD data types from and to files[34]. The next segment of VHDL code illustrates how a simple table of results can be written to file log.txt. The results table is shown in Table 5.8.

```
-- Test textio package.
--
library STD;
use STD.textio.all;
--
entity Qucs_write_test is
end entity Qucs_write_test;
--
```

[33]The specification for the FreeHDL package textio can be found in text file freehdl-0.0.3/std/textio.vhdl.

[34]VHDL allows data to be read from and written to the standard input and output streams as well as user defined files. At this time only writing data to file log.txt and reading data from user defined data files has been tested. Please note that the use of the textio package is very much a cutting edge feature of the Qucs/FreeHDL software and is probably not bug free.

```
architecture behavioural of Qucs_write_test is
begin
write_test: process is
              variable input_line, output_line : line;
              variable int1 : integer := 10;
            begin
                write(output_line, string'("  " ) );
                writeline(output, output_line);
                write(output_line, string'("String -> log.txt" ) );
                writeline(output, output_line);
--
test_L1 :       for  ic in 1 to 5 loop
                  int1 := int1 + 1;
                  write(output_line, string'("int1 =  " )    );
                  write(output_line, int1);
                  write(output_line, string'("   int1^2 =  " )    );
                  write(output_line, int1*int1);
                  writeline(output, output_line);
                end loop test_L1;
                report "Finished test for loop.";
           end process write_test;
end architecture behavioural;
```

```
Output:
----------
Starting new simulation on Thu 24. Aug 2006 at 13:10:56
running C++ conversion... done.
compiling functions... done.
compiling main... done.
linking... done.
simulating...
Output to STD output -> log.txt
int1 =  11   int1^2 =  121
int1 =  12   int1^2 =  144
int1 =  13   int1^2 =  169
int1 =  14   int1^2 =  196
int1 =  15   int1^2 =  225
0 fs + 0d: NOTE: Finished test for loop.
running VCD conversion... done.
Simulation ended on Thu 24. Aug 2006 at 13:10:57
Ready.
Errors:
--------
```

Table 5.8: Log.txt file showing tabular output results.

5.10.8 Testing digital systems using test vectors stored on disk.

In an attempt on my part to review all the new features introduced in the previous sections of this update the final example demonstrates how test vectors stored on disk, as a text file, can be read by the simulation program at the start of a simulation, then applied to the inputs of the digital system under test. The code for this example is given in the following listing:

```
-- Testing digital circuits using test vectors
-- stored as a text file on disk.
--
entity comb1 is
        port (a, b, c, d  : in bit;
                y : out bit
              );
end entity comb1;
--
architecture dataflow of comb1 is
begin
        y <= (a nand b) or (c and d);
end architecture dataflow;
--
library STD;
use STD.textio.all;
--
entity testbench is
end entity testbench;
--
architecture behavioural of testbench is
signal clock : bit;
signal v1, v2, v3, v4, y_out : bit;
type array_list is array (1 to 20) of bit;
signal v1sd, v2sd, v3sd, v4sd  : array_list;
--
Procedure store_data (variable number : out  integer) is
 variable d1, d2, d3, d4 : bit;
 variable in_line, out_line : line;
 variable i  : integer ;
 variable my_string : string(1 to 20) := cr &  "Constrained string" & cr;
 file infile : text open read_mode is "/mnt/hda2/qucs-0.0.10f/test1_data";
begin
 report my_string;
 i := 1;
 while not ( endfile(infile) ) loop
  readline(infile, in_line);
  read(in_line, d4);
```

```
   read(in_line,d3);
   read(in_line,d2);
   read(in_line,d1);
   v1sd(i) <= d1;
   v2sd(i) <= d2;
   v3sd(i) <= d3;
   v4sd(i) <= d4;
   report "In file read loop.";
   i := i+1;
   if (i > 20) then exit;
   end if;
   number:= i;
 end loop;
end procedure store_data;
--
begin
p0 : process is  -- Generate a clock signal.
       begin
        clock <= '1'; wait for 10 ns;
        clock <= '0'; wait for 10 ns;
      end process p0;
--
g0 :  entity work.comb1 port map (v1, v2, v3, v4, y_out);
--
p1 : process is  -- Read test vectors from disk and
--                     apply data to circuit inputs.
       variable no_reads : integer;
       variable in_line, out_line : line;
      begin
       store_data(no_reads);
       write(out_line,string'("count = "));
       write(out_line, no_reads-1);
       writeline(output, out_line);
--
       for k in  1 to no_reads-1 loop -- Count up.
         wait until (clock'event and clock='1');
         v1 <= v1sd(k);
         v2 <= v2sd(k);
         v3 <= v3sd(k);
         v4 <= v4sd(k);
         write(out_line, string'("Time = "),left, 8 );
         write(out_line, now, right, 10);
         write(out_line, string'(" Test vectors -> "),right, 20 );
         write(out_line, v4, left, 2 );
         write(out_line, v3, left, 2 );
```

```
        write(out_line, v2, left, 2);
        write(out_line, v1, left, 2);
        write(out_line, string'("k = "), right, 10 );
        write(out_line,k);
        writeline(output,out_line);
        wait until (clock'event and clock='0');
      end loop;
--
      for k in  no_reads-1 downto 1 loop -- Count down.
        wait until (clock'event and clock='1');
        v1 <= v1sd(k);
        v2 <= v2sd(k);
        v3 <= v3sd(k);
        v4 <= v4sd(k);
        write(out_line, string'("Time = "),left, 8 );
        write(out_line, now, right, 10);
        write(out_line, string'(" Test vectors -> "),right, 20 );
        write(out_line, v4, left, 2 );
        write(out_line, v3, left, 2 );
        write(out_line, v2, left, 2);
        write(out_line, v1, left, 2);
        write(out_line, string'("k = "), right, 10 );
        write(out_line,k);
        writeline(output,out_line);
        wait until (clock'event and clock='0');
       end loop;
      wait;
    end process p1;
end architecture behavioural;
```

Although the listing above is relatively short, careful study of it's contents should allow readers to identify many of the new Qucs/FreeHDL features introduced earlier. Moreover in some sections, the code illustrates extra features which will be familiar to those Qucs/FreeHDL users who have a more advanced knowledge of the VHDL language. These are listed below with a number of general points:

- The VHDL code simulates the performance of a simple combinational logic circuit called comb1: this has four inputs (a, b, c, d) of type bit and one output (y) of type bit[35].

- The testbench being simulated consists of two processes: process p0 generates a clock signal with a period of 20 ns; process p1 inputs test data held in file `test1_data` [36] and stores it in four signal arrays (v1sd, v2sd, v3sd and v4sd), applying this data

[35]Type bit was chosen for this example rather than one of the IEEE signal types because package textio does not handle the IEEE multivalue logic types.

[36]I use the Knoppix version of the Linux/GNU operating system for all work on the Qucs project. The

to the inputs of the circuit under test at the leading edges of the clock pulse. Note process p1 only executes once due to the wait statement at its end.

- An instantiation of the comb1 component is included in the testbench architecture. Note the use of the VHDL *entity work.comb1* construction, this is an alternative for *use work.all*;

- The test vector data held in file `test_data` is read by procedure `store_data` which returns the number of lines of data read in variable number. File handling, including reading data from disk, is undertaken with predefined routines in package textio.

- The first *report* statement in procedure `store_data` writes string `my_string` to file log.txt. `My_string` is an example of the VHDL constrained string type, consisting of non-printable control characters[37] concatenated with printable characters.

- Two loops are employed in process p1 to apply signal test vectors to the input of comb1: the first loop counts up from one and the second loop counts down from the number of lines of test vectors read by procedure `store_data`, effectively generating test vectors in a way similar to using an up-down pattern generator counter. Note that the signal data is applied to the circuit under test on the rising edge of the clock signal and that the applied signal vector sequence is really up to the imagination of the VHDL programmer.

- The write statements in the process p1 for loops demonstrate the formatted version of the textio write statement. This greatly assists in setting up tables of results. Table 5.9 gives a typical log.txt content for the comb1 test simulation.

- In process p1 signals v1, v2, v3 and v4 are assigned an indexed value from (type `array_list`) v1sd, v2sd, v3sd and v4sd signals. During simulation Qucs/FreeHDL stores signal values as a simulation progresses. Hence, it is theoretically possible to display both the standard and composite signal types. A typical waveform plot for signals v1, v2, v3, v4 and `y_out` is given in Fig. 5.30. Fig. 5.31 illustrates a waveform plot of the composite signals v1sd, v2sd, v3sd and v4sd. In Fig. 5.31 each group is plotted at a clock edge change yielding identical groups of values; each vertical set of bits represents the bit values for a single line in file `test1_data`. Compare the displayed values in Fig. 5.31 with the contents of the `test1_data` file shown in Fig. 5.32. As mentioned before some care is needed when plotting, or tabulating, composite signals, particularly when the array sizes are large; array dimensions above roughly 50 become difficult to plot on a normal resolution screen. In such cases it is better to slice part of an array and assign the required values to a signal that can be easily displayed.

absolute location of the test data file will depend on where Qucs and FreeHDL have been installed and the location where work files are kept.

[37] Type character in package standard lists the two letter codes used by VHDL to represent non-printable control characters.

Output:

```
Starting new simulation on Fri 25. Aug 2006 at 14:35:48
running C++ conversion... done.
compiling functions... done.
compiling main... done.
linking... done.
simulating...
0 fs + 0d: NOTE:
Constrained string
0 fs + 0d: NOTE: In file read loop.
.
0 fs + 0d: NOTE: In file read loop.
count =    16
Time =       0 ns   Test vectors ->  0 0 0 0       k = 1
Time =      20 ns   Test vectors ->  0 0 0 0       k = 2
Time =      40 ns   Test vectors ->  0 0 0 1       k = 3
Time =      60 ns   Test vectors ->  0 0 1 0       k = 4
.
Time =     200 ns   Test vectors ->  1 0 0 1       k = 11
Time =     220 ns   Test vectors ->  1 0 1 0       k = 12
Time =     240 ns   Test vectors ->  1 0 1 1       k = 13
Time =     260 ns   Test vectors ->  1 1 0 0       k = 14
Time =     280 ns   Test vectors ->  1 1 0 1       k = 15
Time =     300 ns   Test vectors ->  1 1 1 0       k = 16
Time =     320 ns   Test vectors ->  1 1 1 1       k = 16
Time =     340 ns   Test vectors ->  1 1 1 1       k = 15
Time =     360 ns   Test vectors ->  1 1 1 0       k = 14
Time =     380 ns   Test vectors ->  1 1 0 1       k = 13
Time =     400 ns   Test vectors ->  1 1 0 0       k = 12
.
Time =     560 ns   Test vectors ->  0 1 0 0       k = 4
Time =     580 ns   Test vectors ->  0 0 1 1       k = 3
running VCD conversion... done.
Simulation ended on Fri 25. Aug 2006 at 14:35:50
Ready.
Errors:
```

Table 5.9: An edited version of the formatted tabular output results written to file log.txt.

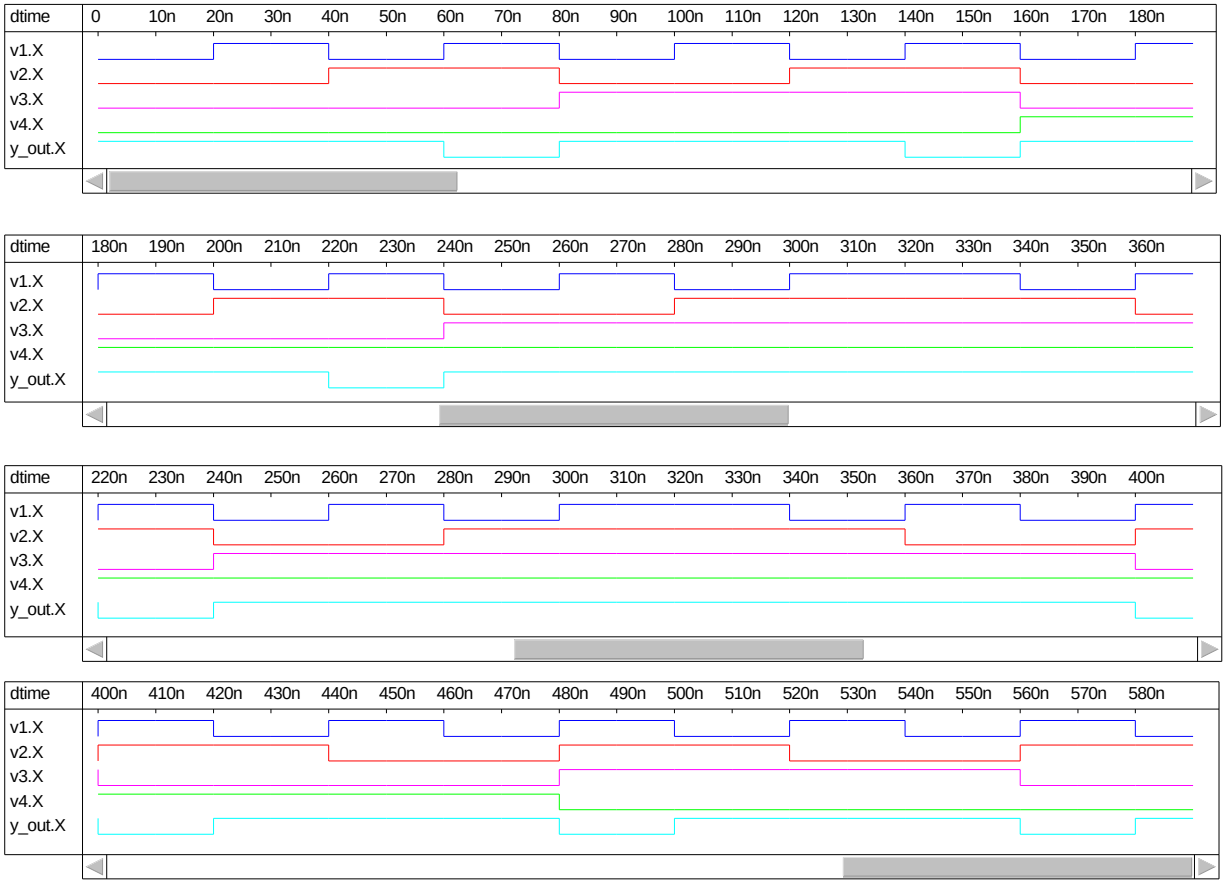


Figure 5.30: Typical timing diagram for comb1 simulation.

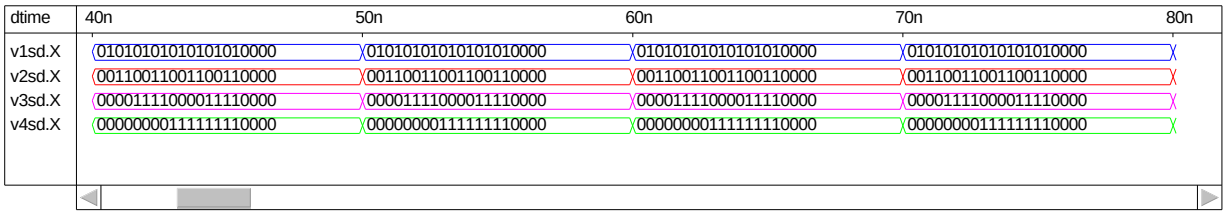


Figure 5.31: Typical timing diagram for composite signals v1sd, v2sd, v3sd and v4sd.

```
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1
```

Figure 5.32: Comb1 simulation test vectors.

5.11 End note

Qucs 0.0.8 added digital simulation to the impressive list of features already available in the Qucs package. The 0.0.8 release represented a significant step forward in the development of the Qucs project. The fact that there were bugs in the first version of the digital simulator was not surprising given the complexity of the software. Release 0.0.9 goes a long way to correcting the most annoying of these bugs. It also adds a number of new features, the most notable being the new VHDL editor and the automatic generation of component symbols from hand crafted VHDL model code. Qucs 0.0.10 and FreeHDL 0.0.3 adds a range of new features to the software, particularly important are the use of the IEEE `std_logic_1164` package and the file handling routines found in the textio package. My thanks to Michael Margraf and Stefan Jahn for all their encouragement during the period that I have been testing the Qucs VHDL digital simulation and the subsequent writing of these notes.

6 Transient Domain Flip-Flop Models for Mixed-Mode Simulation

6.1 Introduction

One of the primary aims of the Qucs project is the development of a universal circuit simulator that allows circuit performance to be investigated from DC to microwave frequencies. Adding performance analysis in the digital domain makes Qucs a truly universal simulator. Qucs 0.0.8 was the first release to include digital simulation. Qucs digital simulation centres around VHDL using the FreeHDL VHDL compiler to generate a machine code simulation of a circuit under test. Release 0.0.8 includes built-in models for the basic digital gates and a number of the common sequential flip-flops. The Qucs gate models can be used in both digital and transient simulation. Unfortunately, the flip-flop models are only available in digital simulation. The current version of Qucs models flip-flops using VHDL and does not provide time domain models for transient simulation. This is an important omission which limits the Qucs simulator mixed-mode simulation capabilities. Mixed-mode simulation is a term commonly employed to describe the simulation of circuits that contain both analogue and digital components. In the real world circuits are, of course, not subdivided into neat boxes labelled analogue, S-parameter, digital or any other physical domain. So it is of some importance that Qucs device modelling be developed to allow circuits consisting of a range of different analogue and digital components, to be simulated at the same time. Normally such systems are simulated in the time domain using large signal transient simulation. Performance data being both analogue and digital expressed in tabular or graphical form. This tutorial note presents a number of transient simulation models for flip-flops based on structural digital circuits, describes their use, and outlines a number of example simulations derived from practical circuits.

6.2 Latches and flip-flops

Sequential digital devices generically known as flip-flops (SR, D, JK and T types) are commonly classified into three major groups.

- Latches: basic or gated
- Pulse triggered flip-flops: master slave devices with or without data-lockout
- Edge-triggered flip-flops: leading or trailing edge triggered.

As the speed of electronic systems has increased so has the popularity of the single edge-triggered flip-flops over the slower master slave devices. Today most IC designs are based on D type edge-triggered devices rather than the earlier JK master slave devices. Our concern here is the development of a consistant set of models that allow the common flip-flops to be modelled accurately, and reliably, in the transient time domain. In order to keep these models simple the D gated and edge-triggered devices have been chosen as the fundamental building blocks for the transient domain Qucs models. Using basic Boolean logic concepts it is straightforward to show that JK and T edge-triggered flip-flop models can be derived from the D flip-flop models.

6.3 The gated D latch

The circuit diagram for a gated D latch constructed from two input nand gates is shown in Fig. 6.1[1]. Outputs Q and not Q (QB in Fig. 6.1) are derived from the two cross coupled nand gates connected as a basic SR nand latch. Fig. 6.2 shows the performance characteristics for this circuit. These were obtained using the simple test configuration shown in Fig. 6.3. Logic one digital signals are represented by 1V and logic 0 signals by 0V in the transient analysis domain. Propagation delays through the various circuit gates can be set by changing the delay time for each gate. Cross coupled gates are often a cause of simulation failure due to the fact that DC analysis fails to converge to a stable solution at the start of a transient simulation. One approach that helps to force a stable DC solution is to set Q and QB to known values, say logic 0 and logic 1, at the start of a simulation. In circuits like the basic gated D latch shown in Fig. 6.1, where asynchronous set and reset inputs are not included, this is not possible. However, flip-flops with asynchronous set and reset inputs do allow the state of a flip-flop to be determined at a given time in a simulation. In the examples that follow, whenever possible, the state of the latch or flip-flop devices is set at the start of a simulation. In the majority of the example circuits, device delays have also been set to zero. It therefore follows that most waveform plots show functional data rather than accurate timing characteristics. In many mixed-mode simulations the digital elements present in a design are often modelled as functional devices whose primary task is to generate the signals needed for the overall circuit to function. A more detailed discussion of the effects on transient simulation caused by including device timing delays is presented in a later section of these notes.

[1]Richard S. Sandige, Modern Digital Design, 1990, McGraw-Hill International Editions.

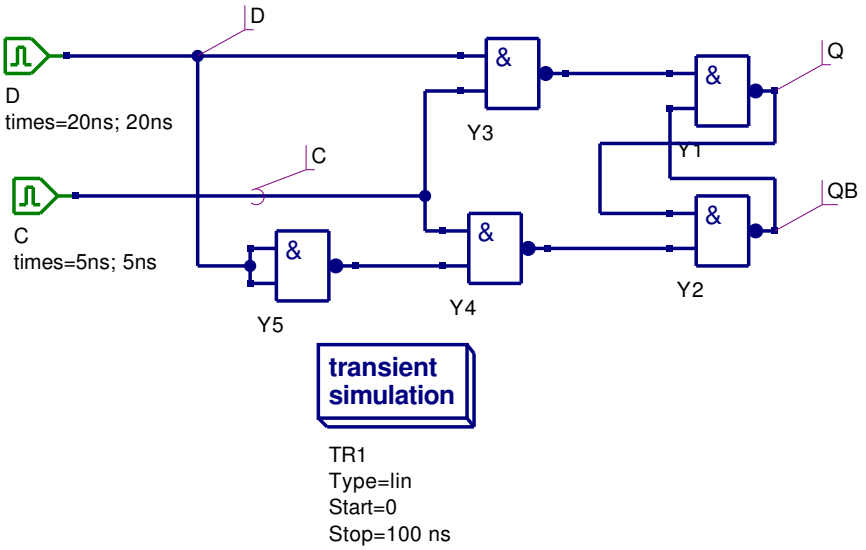


Figure 6.1: Gated D latch with digital signal generators D and C

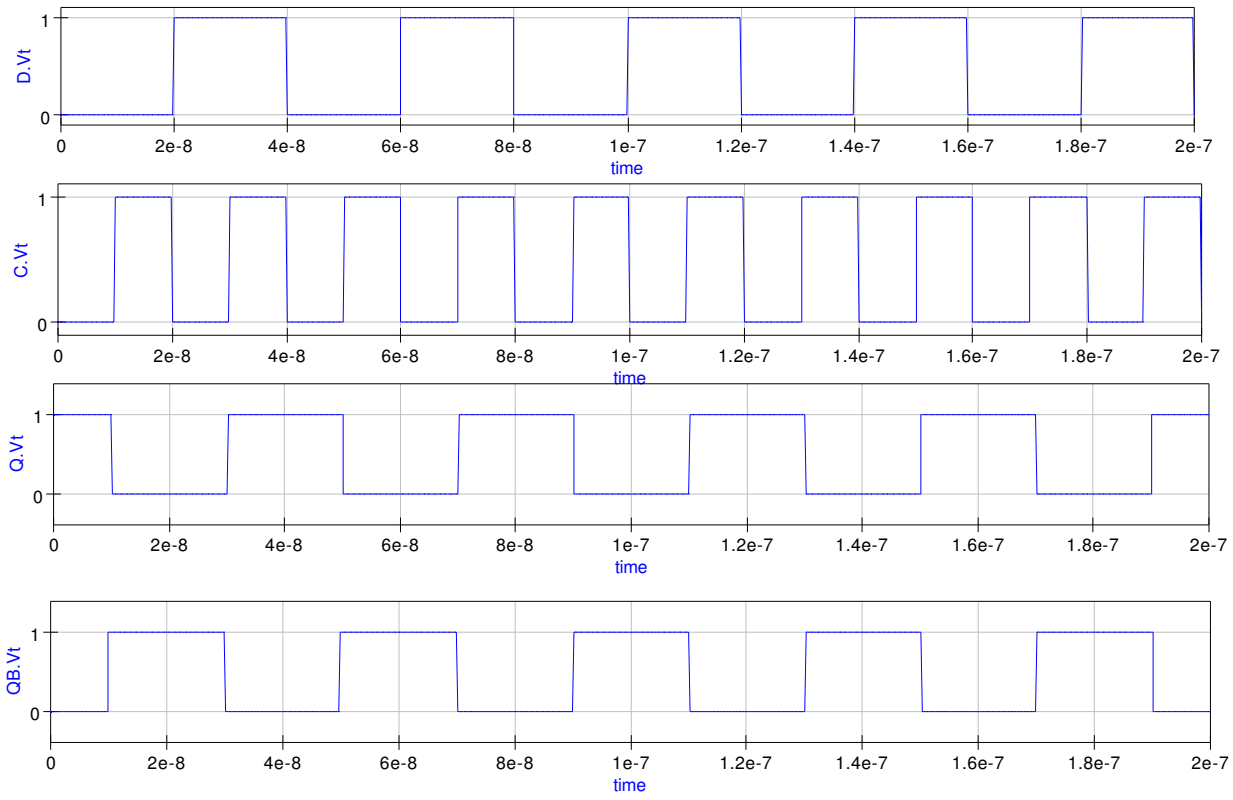


Figure 6.2: Gated D latch simulation waveforms

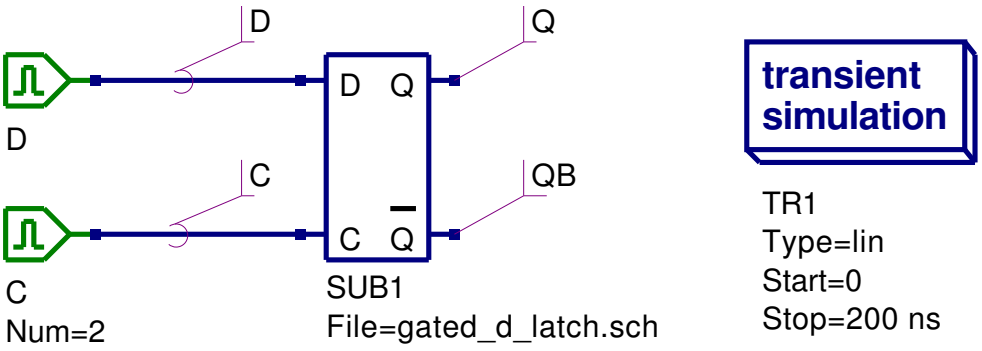


Figure 6.3: Gated D latch test circuit

6.4 Edge-triggered D type flip-flop

The schematic for a positive edge-triggered D flip-flop is shown in Fig. 6.4[2]. Asynchronous set (SET) and reset (RESET) control inputs allow the flip-flop outputs Q and not Q (QB in Fig. 6.4) to be set to known values at the start of a simulation. The nand gates forming each of the cross coupled SR latches have their delay times set at 0 ns. The edge-triggered D device is a building block for both the JK and T types of flip-flop. A typical set of transient simulation test results for the D flip-flop model are illustrated in Fig. 6.5. These where obtained using the basic test configuration shown in Fig. 6.6.

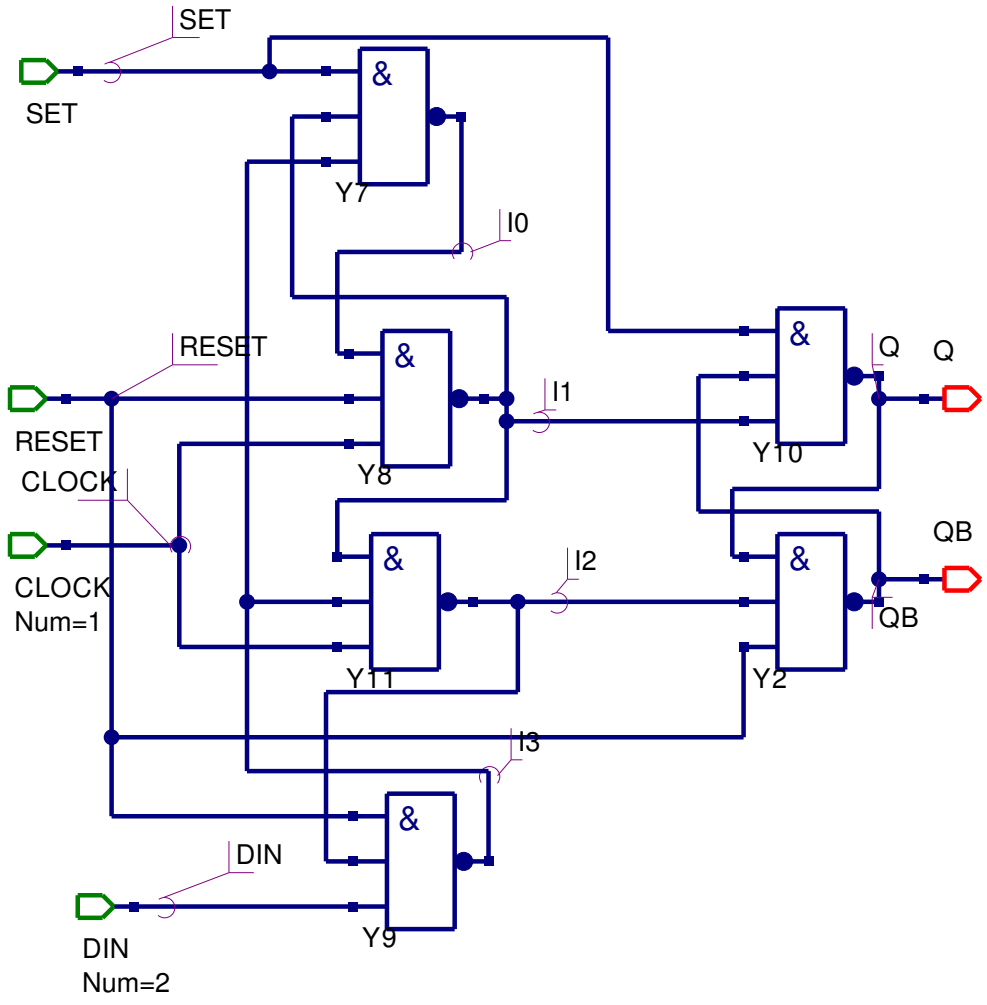


Figure 6.4: Positive edge-triggered D flip-flop circuit

[2]David A. Hodges and Horace G. Jackson, Analysis and Design of Digital Integrated Circuits, 1998, Second edition, McGraw-Hill Book Company.

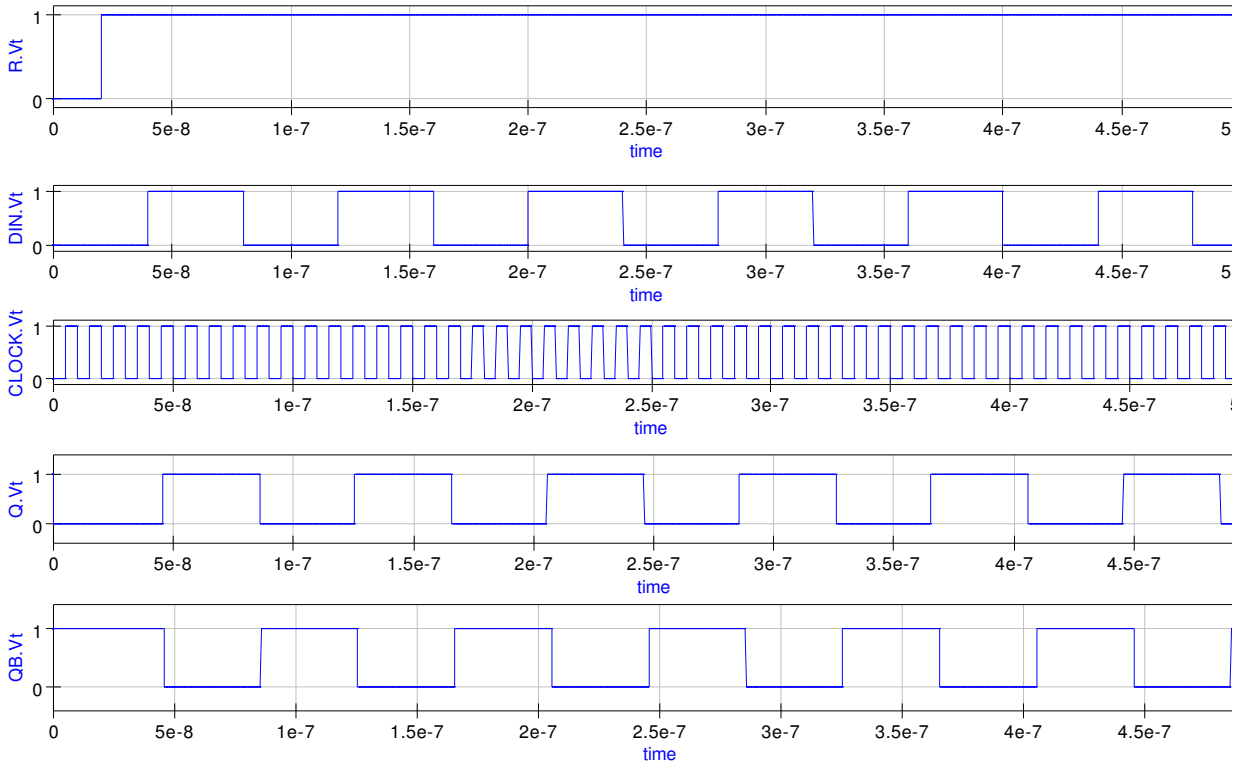


Figure 6.5: Transient waveforms for the circuit shown in Fig. 6.6

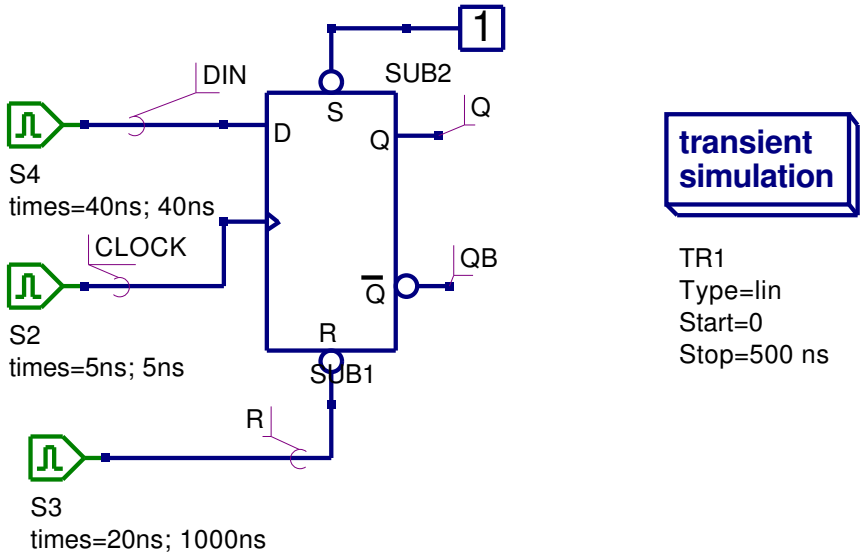


Figure 6.6: D flip-flop test circuit

6.5 The edge-triggered JK flip-flop

A leading edge-triggered JK flip-flop can be constructed using a positive edge-triggered D flip-flop and external logic[3]. The external logic generates the required JK flip-flop characteristic equation given by

$$Q^+ = J.\overline{Q} + \overline{K}.Q$$

Were Q, $\overline{Q}$, J and $\overline{K}$ are the current state values of the device signals and Q^+ is the next state value of Q following the rising edge of the device clock pulse. The schematic diagram for the edge triggered flip flop is shown in Fig. 6.7 and a typical set of test waveforms in Fig. 6.8. These were obtained using the test circuit shown in Fig. 6.9.

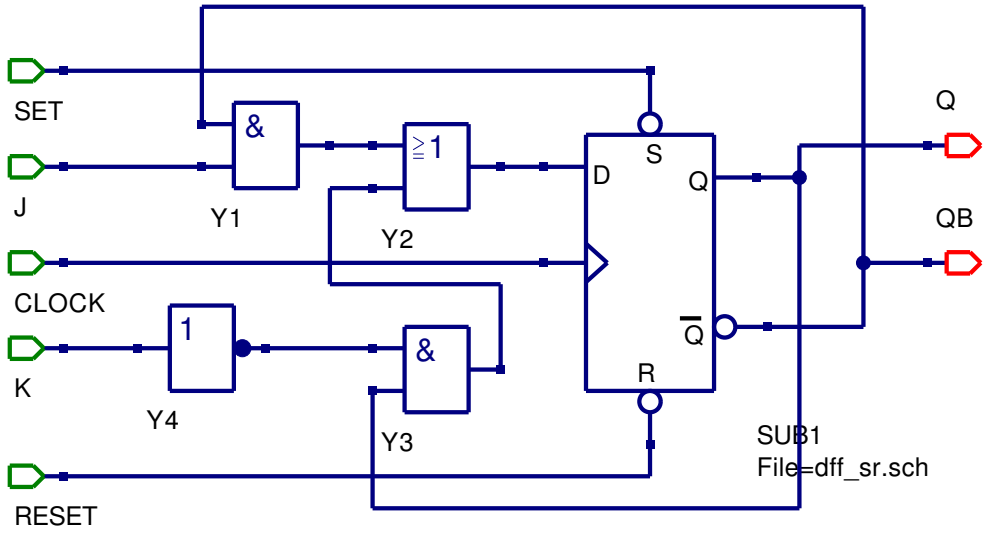


Figure 6.7: Positive edge-triggered JK flip-flop circuit

[3]M. Morris Mano and Charles R Kime, Logic and Computer Design Fundamentals, 2004, Third edition, Pearson Education International, Prentice Hall

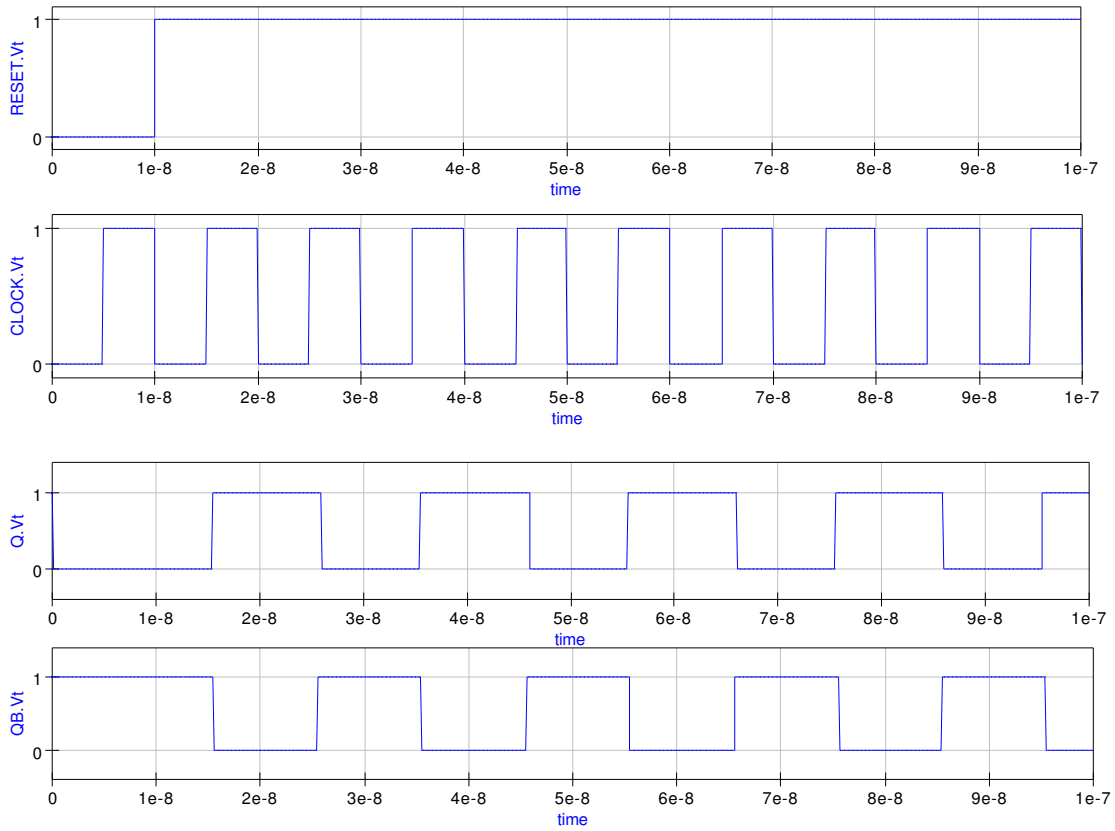


Figure 6.8: Transient waveforms for the circuit shown in Fig. 6.9

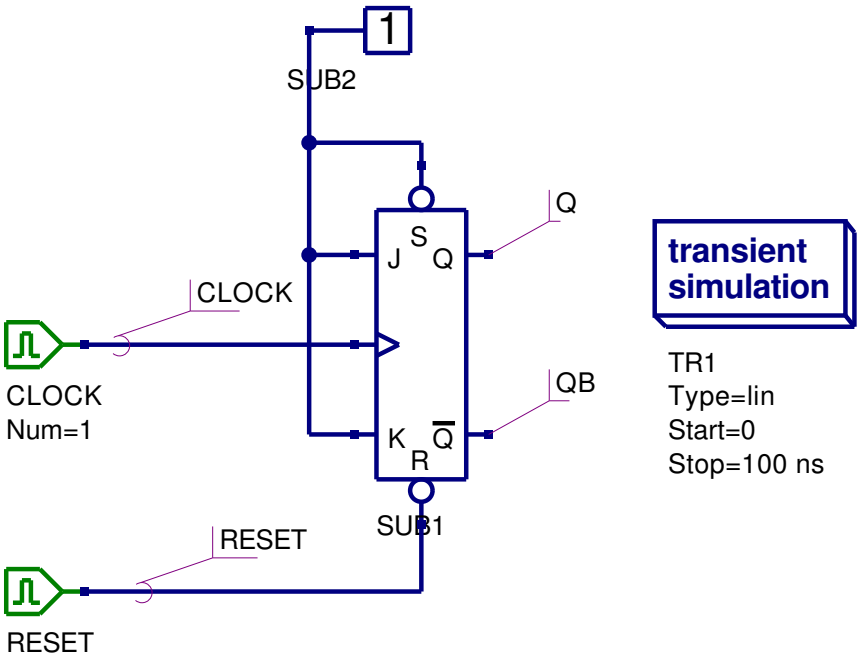


Figure 6.9: JK flip-flop test circuit showing JK operating in toggle mode

6.6 The edge-triggered T flip-flop

The characteristic equation for a leading edge-triggered flip-flop is[4]

$$Q^+ = T \oplus Q$$

where the symbols have the same meaning as the JK flip-flop. The circuit diagram, test waveforms and test circuit for the edge-triggered flip-flop are given in Figures 6.10 to 6.12.

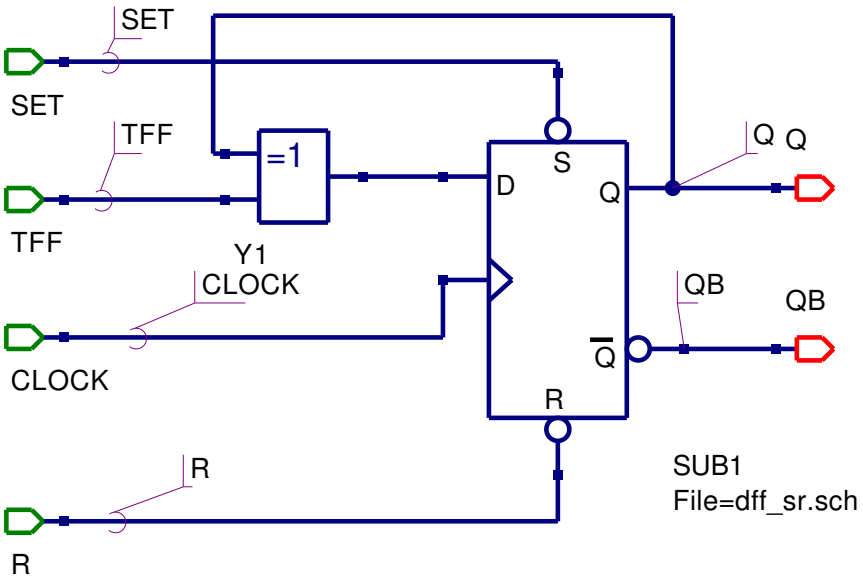


Figure 6.10: Positive edge-triggered T flip-flop circuit

[4]See footnote 2.

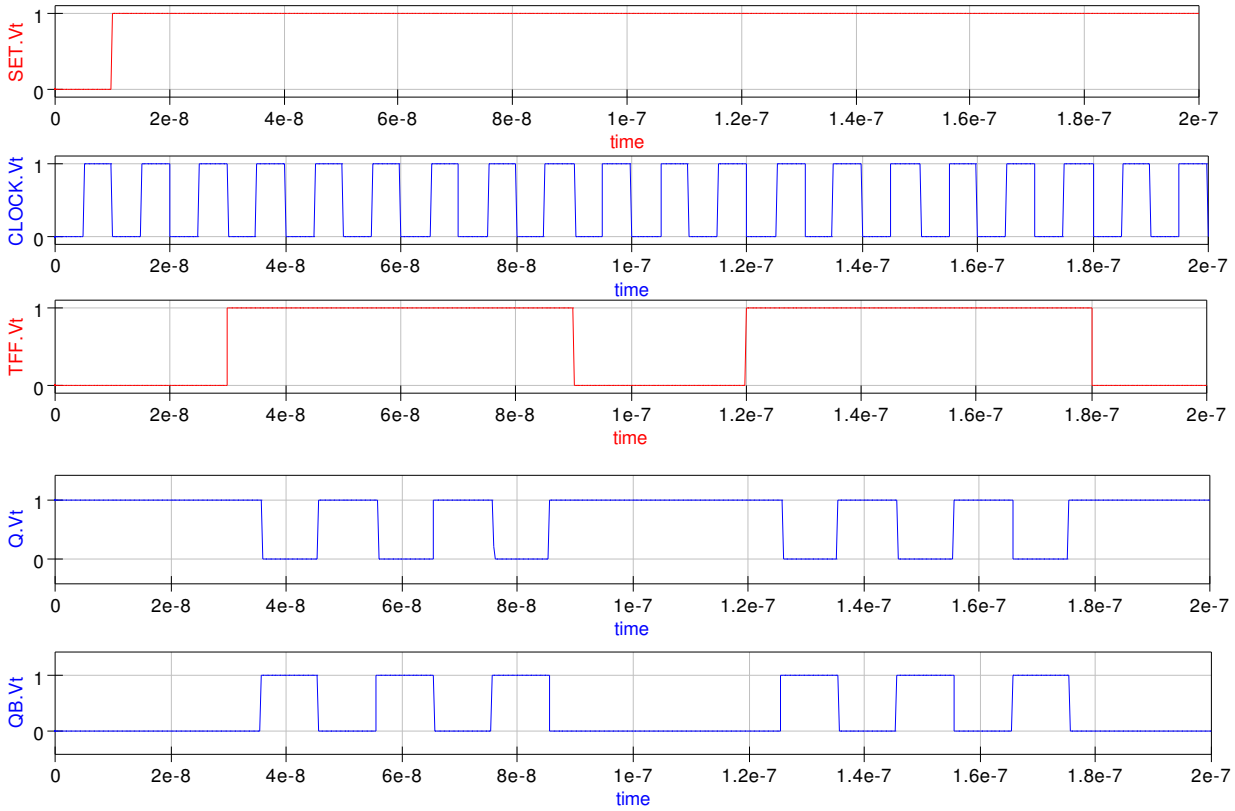


Figure 6.11: Transient waveforms for the circuit shown in Fig. 6.12

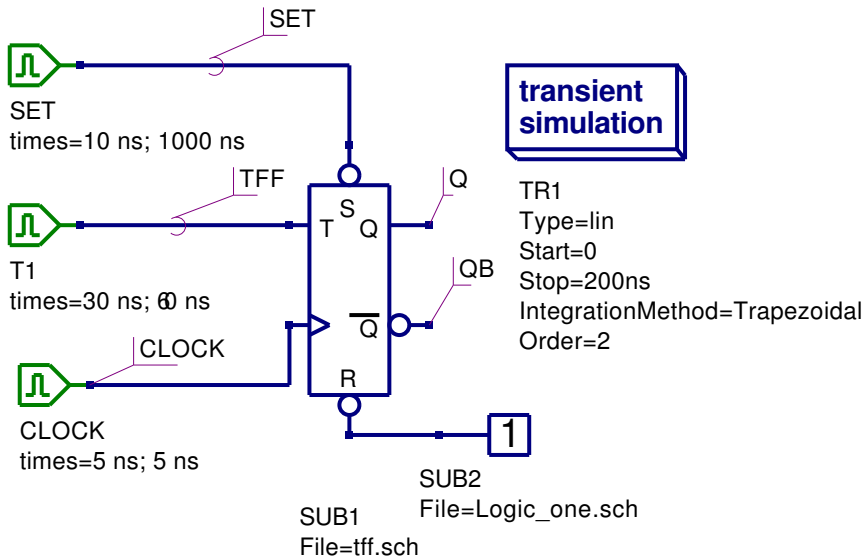


Figure 6.12: T flip-flop test circuit

6.7 Two example digital circuits

- **A synchronous BCD up-counter**: Figure 6.13 shows a synchronous BCD up-counter constructed from four edge-triggered JK flip flops connected as toggle flip-flops. The input signal waveforms and the corresponding counter outputs Q0, Q1, Q2 and Q3 are illustrated in Fig. 6.14. These simulation results were obtained using the default trapezoidal integration method with order 2.

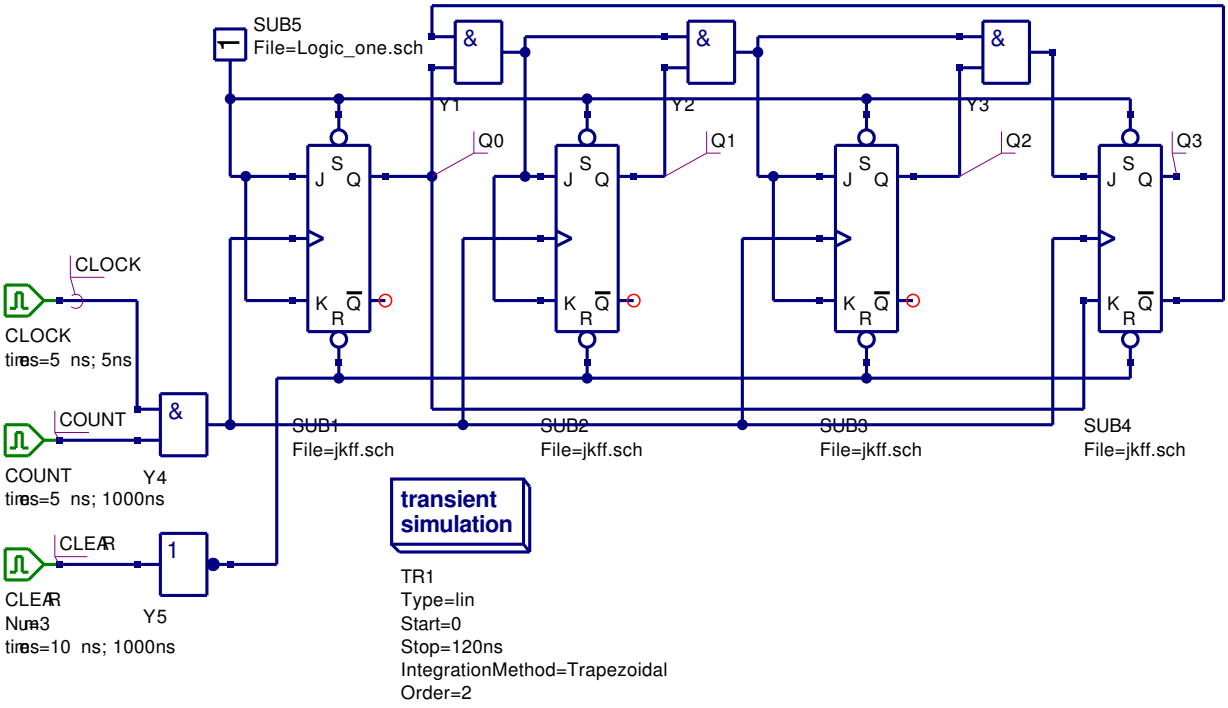


Figure 6.13: Synchronous BCD up-counter circuit

At the start of simulation signal CLEAR is set to logic 1 which in turn causes the counter to be reset to 0000. Similarly signal COUNT has to be set to 1 for counting to take place. Notice that the counter counts from 0 to 9 and then resets to 0.

- **A simple state machine**: Figure 6.15 shows a simple sequential state machine with input X and outputs Y1 and Y2. The outputs are synchronised to the input clock. The state equations for this example are

$$J = \overline{X},\ K = 1,\ Y1 = \overline{Q0}.\overline{X},\ Y2 = Q0$$

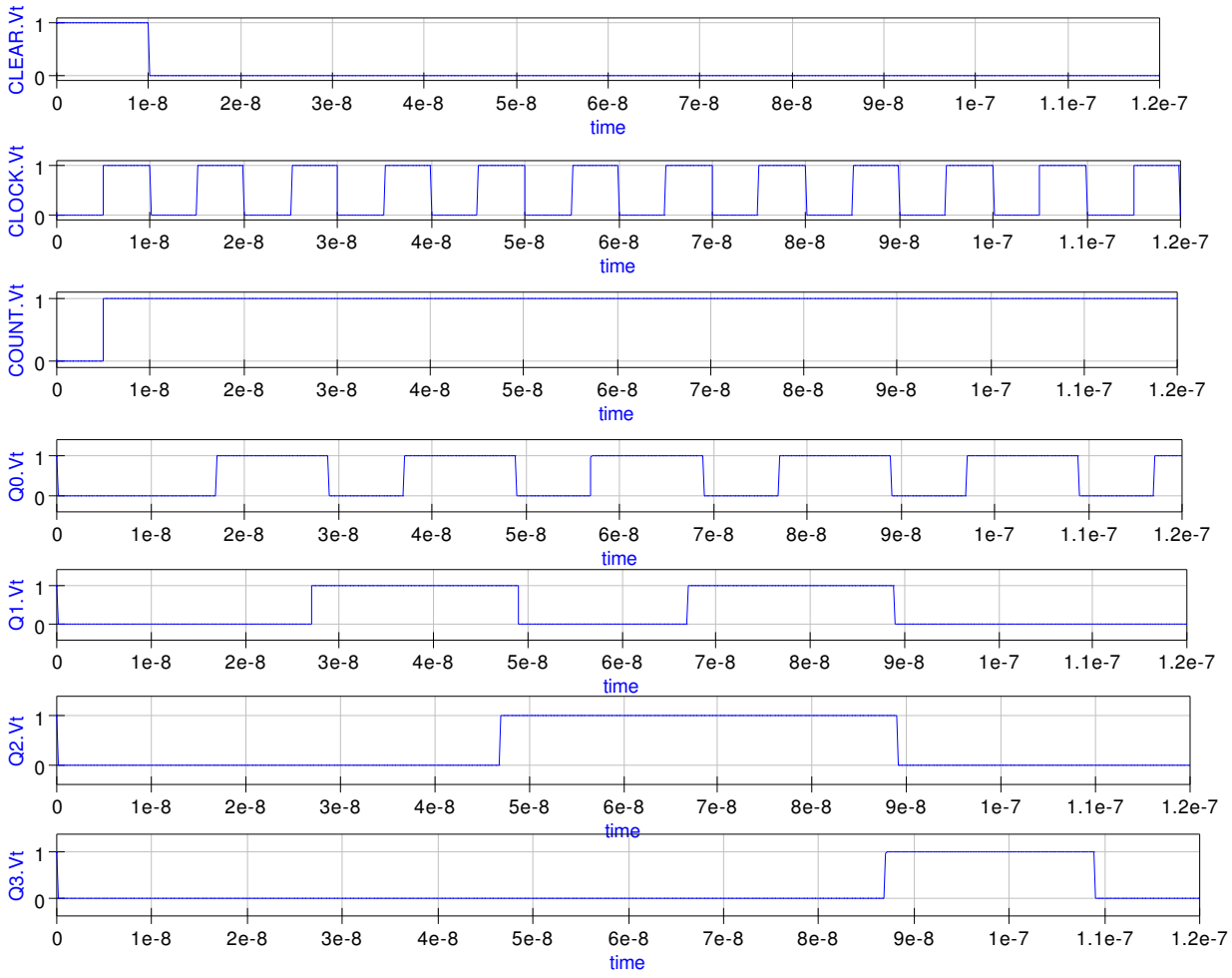


Figure 6.14: Transient waveforms for the circuit shown in Fig. 6.13

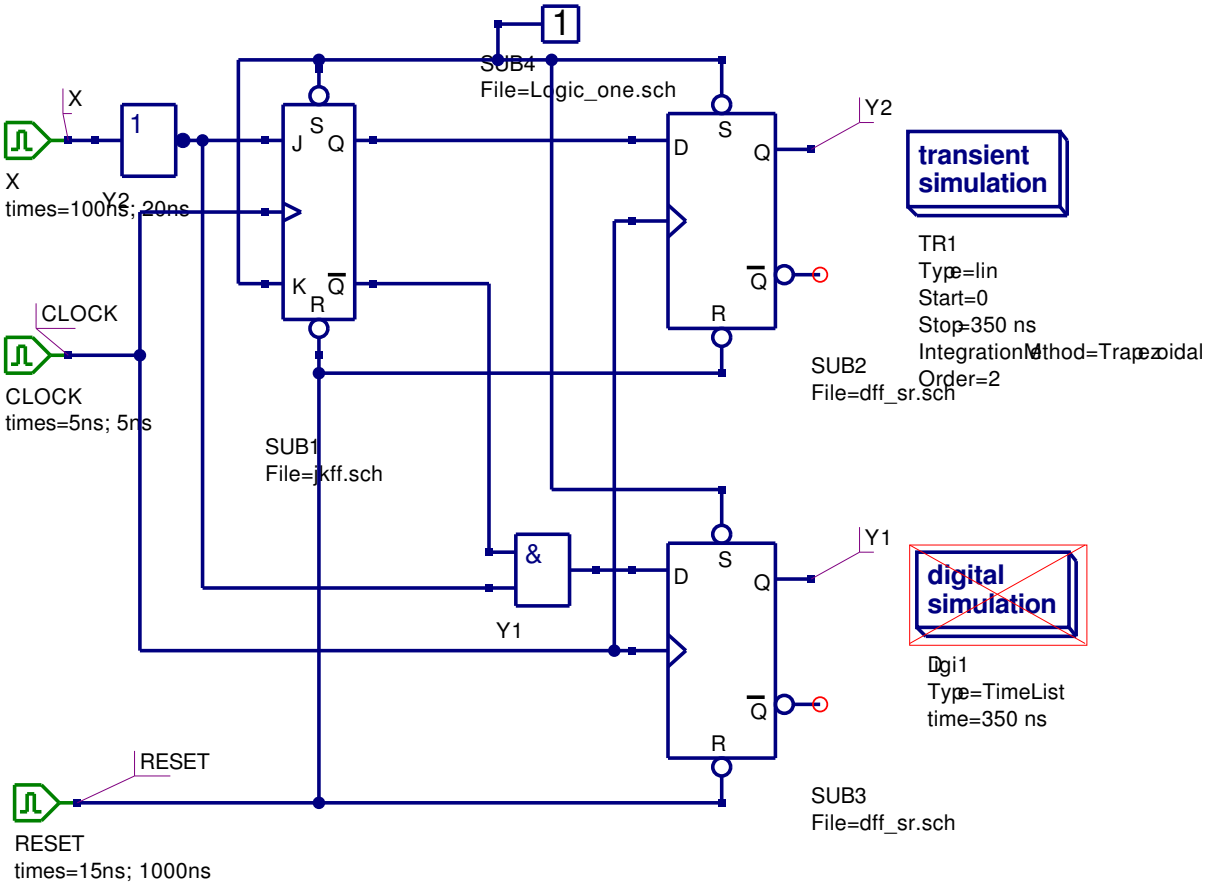


Figure 6.15: A simple state machine

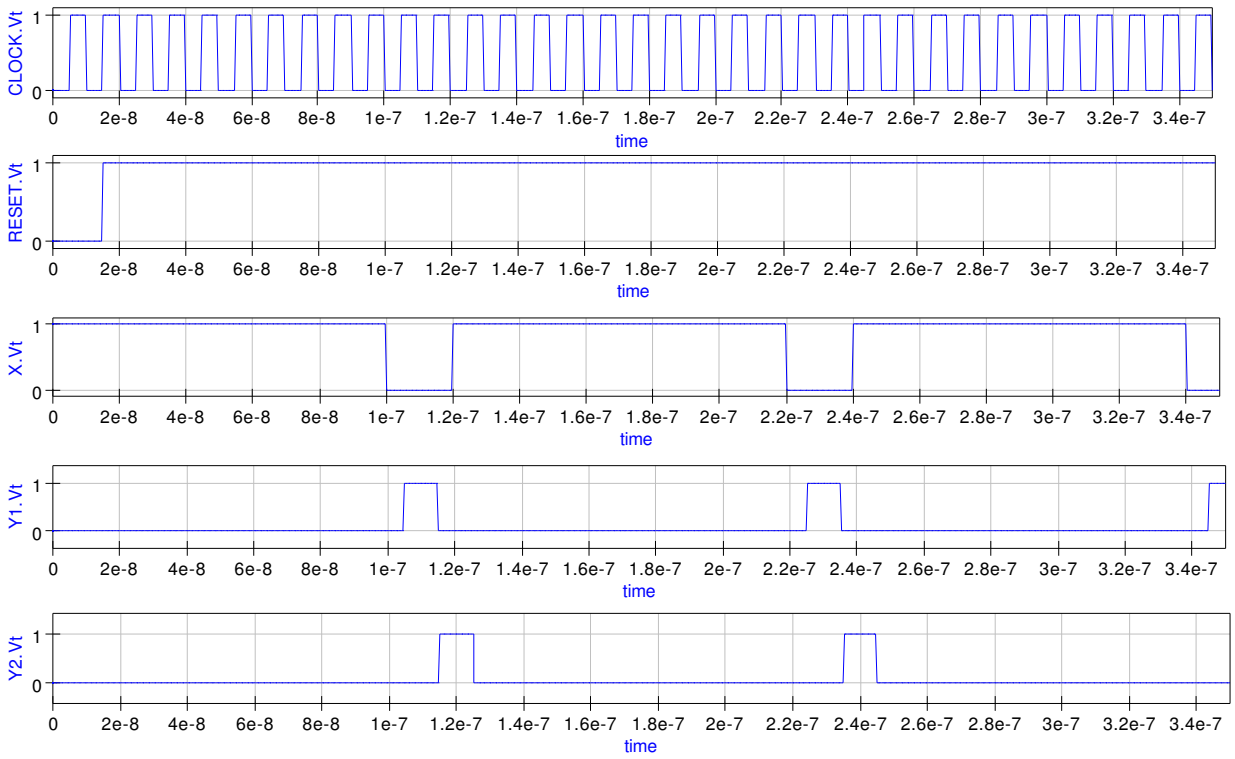


Figure 6.16: Transient waveforms for the circuit shown in Fig. 6.15

6.8 VHDL code for the transient domain flip-flop models

Although the primary purpose for developing the transient domain flip-flop models is the simulation of mixed-mode circuits, it is worth noting that because the models have been constructed from Qucs gate primitives using a bottom-up design approach, Qucs can also use the models for digital simulation. Moreover, provided the circuit being simulated does not contain any purely analogue components Qucs will generate a VHDL model testbench that describes the function and test sequence for the circuit being simulated. Shown in Fig. 6.17 is a digital timelist waveform plot for the synchronous BCD up-counter introduced in the previous section of these notes. Listing 6.1 lists the VHDL code generated by Qucs for the synchronous BCD up-counter example.

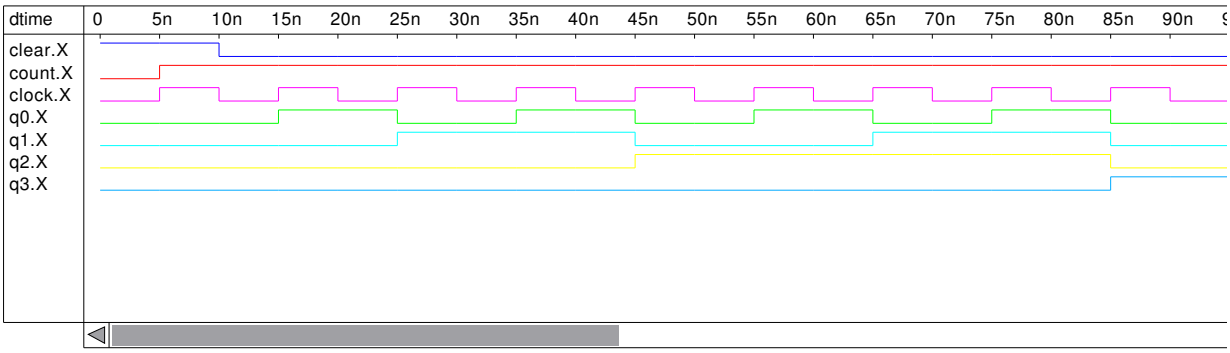


Figure 6.17: Digital TimeList waveforms for the circuit shown in Fig. 6.13

Listing 6.1: VHDL testbench code for the circuit shown in Fig. 6.13

```
-- Qucs 0.0.9
-- /mnt/hda2/Digital_Subcircuits_prj/Sync_BCD_counter.sch

entity Sub_Logic_one is
  port (nnout_L1: out bit);
end entity;
use work.all;
architecture Arch_Sub_Logic_one of Sub_Logic_one is
  signal gnd,
         L1 : bit;
begin
  gnd <= '0';
  L1 <= not gnd;
  nnout_L1 <= L1 or '0';
end architecture;
```

```
entity Sub_dff_sr is
  port (CLOCK: in bit;
        DIN: in bit;
        nnout_Q: out bit;
        nnout_QB: out bit;
        RESET: in bit;
        SET: in bit);
end entity;
use work.all;
architecture Arch_Sub_dff_sr of Sub_dff_sr is
  signal I0,
         I2,
         I1,
         I3,
         QB,
         Q : bit;
begin
  nnout_QB <= QB or '0';
  nnout_Q <= Q or '0';
  I1 <= not (CLOCK and RESET and I0);
  I3 <= not (DIN and I2 and RESET);
  QB <= not (RESET and I2 and Q);
  Q <= not (I1 and QB and SET);
  I0 <= not (I3 and I1 and SET);
  I2 <= not (CLOCK and I3 and I1);
end architecture;


entity Sub_jkff is
  port (nnnet6: in bit;
        nnnet1: in bit;
        nnnet8: in bit;
        nnout_nnnet3: out bit;
        nnout_nnnet7: out bit;
        nnnet9: in bit;
        nnnet10: in bit);
end entity;
use work.all;
architecture Arch_Sub_jkff of Sub_jkff is
  signal nnnet0,
         nnnet2,
         nnnet4,
         nnnet5,
```

```
          nnnet7,
          nnnet3 : bit;
begin
  nnnet0 <= not nnnet1;
  nnnet2 <= nnnet3 and nnnet0;
  nnnet4 <= nnnet2 or nnnet5;
  nnnet5 <= nnnet6 and nnnet7;
  nnout_nnnet7 <= nnnet7 or '0';
  nnout_nnnet3 <= nnnet3 or '0';
  SUB1: entity Sub_dff_sr port map (nnnet8, nnnet4, nnnet3,
                                    nnnet7, nnnet10, nnnet9);
end architecture;

entity TestBench is
end entity;
use work.all;

architecture Arch_TestBench of TestBench is
  signal CLEAR,
         COUNT,
         CLOCK,
         Q3,
         Q0,
         Q1,
         Q2,
         nnnet0,
         nnnet1,
         nnnet2,
         nnnet3,
         nnnet4,
         nnnet5,
         nnnet6,
         nnnet7,
         nnnet8,
         nnnet9 : bit;
begin
  SUB5: entity Sub_Logic_one port map (nnnet0);
  nnnet1 <= Q0 and nnnet2;
  nnnet3 <= Q1 and nnnet1;
  nnnet4 <= Q2 and nnnet3;
  SUB2: entity Sub_jkff port map (nnnet1, nnnet1, nnnet5,
                                  Q1, nnnet6, nnnet0, nnnet7);
```

```
  SUB3: entity Sub_jkff port map (nnnet3, nnnet3, nnnet5,
                                  Q2, nnnet8, nnnet0, nnnet7);
  SUB1: entity Sub_jkff port map (nnnet0, nnnet0, nnnet5,
                                  Q0, nnnet9, nnnet0, nnnet7);
  nnnet5 <= COUNT and CLOCK;
  nnnet7 <= not CLEAR;

  CLEAR: process
  begin
    CLEAR <= '1';  wait for 10 ns;
    CLEAR <= '0';  wait for 1000 ns;
  end process;


  COUNT: process
  begin
    COUNT <= '0';  wait for 5 ns;
    COUNT <= '1';  wait for 1000 ns;
  end process;


  CLOCK: process
  begin
    CLOCK <= '0';  wait for 5 ns;
    CLOCK <= '1';  wait for 5 ns;
  end process;

  SUB4: entity Sub_jkff port map (nnnet4, Q0, nnnet5,
                                  Q3, nnnet2, nnnet0, nnnet7);
end architecture;
```

6.9 Generating a library of mixed-mode digital components

The Qucs project facilities offer users a simple and convenient approach to developing libraries of components that are linked by a common theme; in these notes this is digital component models for transient simulation. To form a library create a new folder, at a point on a disk file system that users have read/write access, giving it a suitable name, for example

```
flip_flop_models_tran_sim_prj.
```

Next move into the new library folder a copy of each of the schematic capture files for the flip-flop models introduced in these notes. These are:

dff_sr.sch, jkff.sch, tff.sch, and gated_d_latch.sch.

A copy of the schematic for setting nodes to logic one is also required

(logic_one.sch).

These models are then freely available for use in any projects which users are working on. They can be copied into such projects using the "Add files to Project..." menu button found under the Qucs Project drop-down menu. Similarly any new models developed as part of a project can be added to the library and used again in the future.

6.10 Digital component propagation time delays and transient simulation numerical stability

Transient simulation is in general much slower than digital simulation using VHDL generated machine code. The large signal transient simulation models of flip-flops and other sequential digital devices are intended for use in mixed-mode circuit simulation rather than being used for pure digital circuit simulation. An interesting, and indeed very important question, relates to the efficiency, and accuracy, of the numerical analysis algorithms employed in the integration routines that are central to transient circuit simulation. Qucs allows users to select the algorithm they wish to employ for transient simulation. The available algorithms are Trapezoidal, Euler, Gear and Adams Moulton; in each case the algorithm order can be set from 1 to 6. The second order Trapezoidal integration algorithm is used by Qucs as the default for transient simulation. To test which of these algorithms offers the most time efficient solution to the transient simulation of digital circuits, that include flip-flops, the BCD counter shown in Fig. 6.13 was used as a test case and repeatedly simulated using different integration routines and algorithm orders. The test results are shown in Table 6.1. Very little difference was found between circuits where the cross coupled gates both had zero propagation delays and the case where one gate had 0.5ns delay and the other zero delay.

One obvious fact emerges from the data given in Table 6.1; namely that the Adams Moulton

Order	Trapezoidal	Euler	Gear	Adams Moulton
1	1	1.62	1.65	1.62
2	1	1.62	0.44	1
4	1	1.62	1.28	0.39
6	1	1.62	0.28	0.18

Table 6.1: Relative simulation times for the circuit shown in Fig. 6.13

Order	Number or rejections	Average time step
1	1470	5.17737e-12
2	1750	9.4585e-12
4	1454	2.866e-11
6	61	5.76646e-11

Table 6.2: Number of rejections and average time step data for the Adams Moulton algorithm

higher order integration routines appear to be faster than the default trapezoidal algorithm. This is corroborated by the average time step and number of rejection data points output by Qucs at the end of a simulation. Table 6.2 lists this data for the Adams Moulton algorithm tabulated in Table 6.1.

Table 6.2 points to the increase in average time step and the dramatic reduction in the number of simulation solution rejections as the probable reason for the reduction in transient simulation time when using the higher order Adams Moulton integration routines. However, other factors may influence the choice of integration routine. Often speed is not the only criteria that is of importance when simulating large complex circuits. Consider the following case (the circuit shown in Fig. 6.13 with order 6 Adams Moulton transient analysis integration); setting one of the gate delays to 1ns, and the other to 0ns, in each of the RS latches in the edge-triggered D flip-flop yields the signal waveforms illustrated in Fig. 6.18. Clearly here the solution is incorrect pointing to probable numerical instability caused by the choice of integration routine.

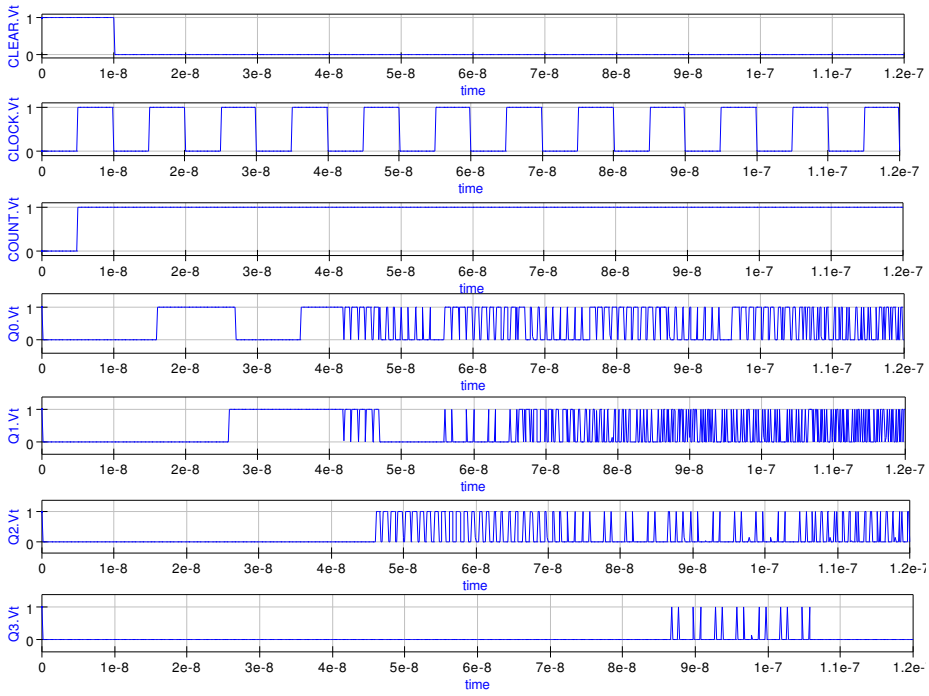


Figure 6.18: Digital TimeList waveforms for the circuit shown in Fig. 6.13

6.11 Mixed-mode example simulations

Mixed-mode simulation involves the simulation of circuits that contain electronic devices and circuits from different physical domains; the most obvious being circuits with a mixture of analogue and digital components. Qucs has developed to a point where it can handle this type of circuit given device models that can span across the different physical domains. In the future such circuits are likely to incorporate components from other domains, including for example, digital signal processing components (DSP) and possibly nano mechanical devices. Multi-domain simulation adds additional complexity to the simulation process not normally found in single domain simulation. Each domain usually represents signal data in a specific way attributed to a given domain; voltage and current for analogue quantities, boolean '1' and '0' for digital signals and floating point numbers for DSP. Hence, signals passing from one domain to another have to be converted from one format to another. These conversion elements are often called node-bridges and form an essential part of the mixed-mode simulation process. The three examples that are introduced in this section of these notes have been chosen to illustrate a number of the basic ideas concerned with mixed-mode simulation of circuits containing analogue and digital components, and to show how Qucs deals with this type of simulation. In the last section the importance of correct selection of integration routine when simulating circuits in the time domain was stressed. Mixed-mode circuits often include a wide diversity of components that exhibit widely differing time constants. This makes the problem of numerical stability versus simulation run time even more critical. With the explicit numerical integration routines,

like the trapezoidal routine, numerical instability results if the simulation time step becomes much larger than the smallest time constant in a circuit. Hence, to achieve successful completion of a simulation the integration time step must be reduced which in turn makes the overall simulation time increase significantly. The implicit Gear algorithm[5] does not suffer from this problem and is the natural choice for circuits with components that have widely differing time constants.

- Example 1: Analogue waveform driven digital devices with output node-bridge.

  The circuit in Fig. 6.19 shows an analogue voltage source driving a digital inverter with a node-bridge element processing the inverter output signal. The input signal is a sinusoidal voltage of amplitude 1V peak. The inverter output signal, V1 in Fig. 6.19, has an nonsymmetrical mark to space ratio because the threshold point for the inverter is set at 0.5V; the halfway point for the two logic levels. The node-bridge element is basically a voltage controlled voltage source where the device gain and time delay can be programmed. In this first example the gain has been set to 5 and the time delay to 0.5ns. Figure 6.20 illustrates the simulation TimeList waveforms for this example mixed-mode circuit. The node-bridge shown in Fig. 6.19 is a very basic device. Moreover, by adding additional features, parameters like fall and rise time can set to specific values. The next example demonstrates the use of an active node-bridge.

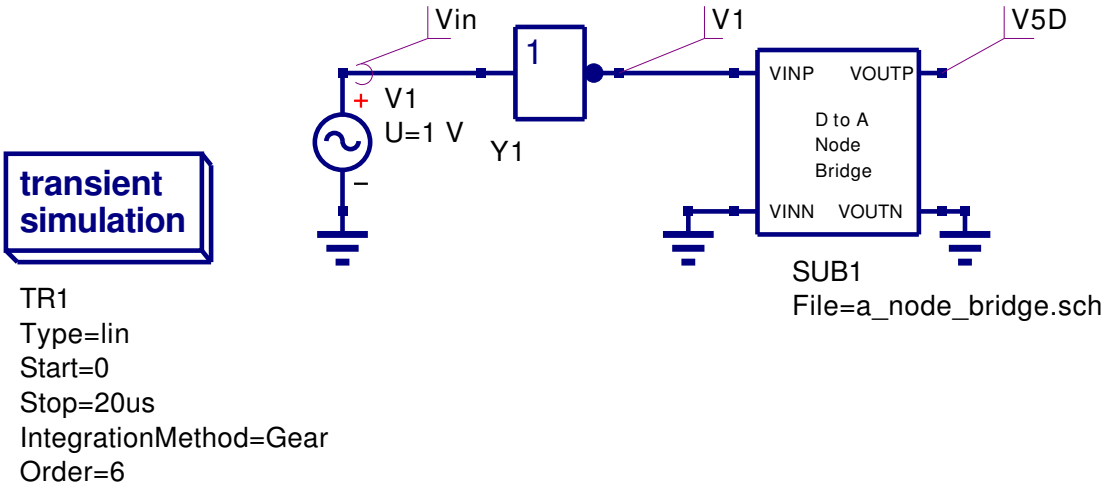


Figure 6.19: Analogue waveform driven digital device with output node-bridge

[5]The Gear integration algorithm is a powerful method for solving stiff systems of differential equations, see Donald A. Calahan, Computer Aided Network Design, Revised edition, 1972, McGraw-Hill.

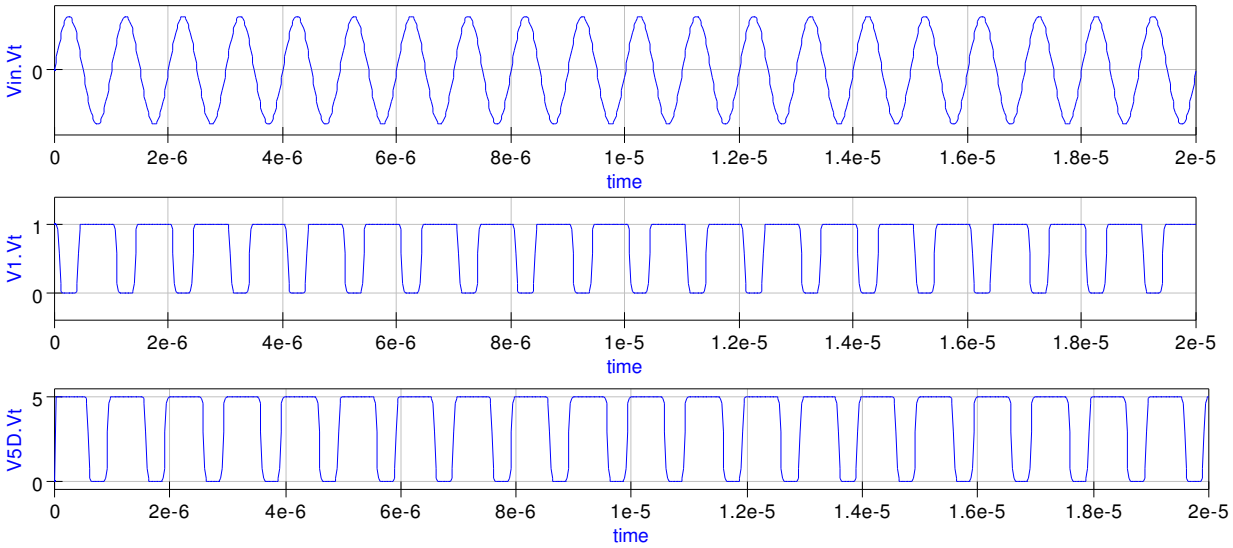


Figure 6.20: Digital TimeList waveforms for the circuit shown in Fig. 6.19

- Example 2: Pulse driven digital inverter with an active node bridge.

  Illustrated in Fig. 6.21 is a similar circuit to the previous example. In Fig. 6.21 a pulse generator drives a digital inverter. The inverter output signal is processed by an active node-bridge derived from a basic BJT switching amplifier. The output waveforms for this circuit are shown in Fig. 6.22. Notice that the pulse rise and fall times are determined by the node-bridge amplifier and that the resulting analogue signal amplitude is set to 5V.

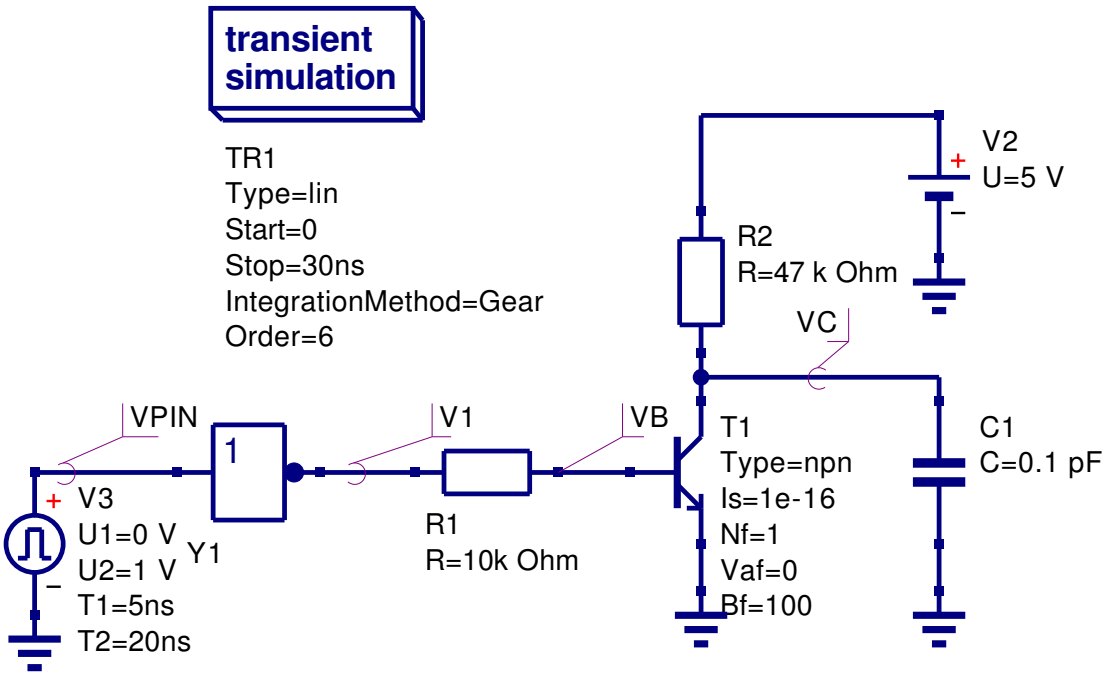


Figure 6.21: Pulse driven digital inverter with active node-bridge

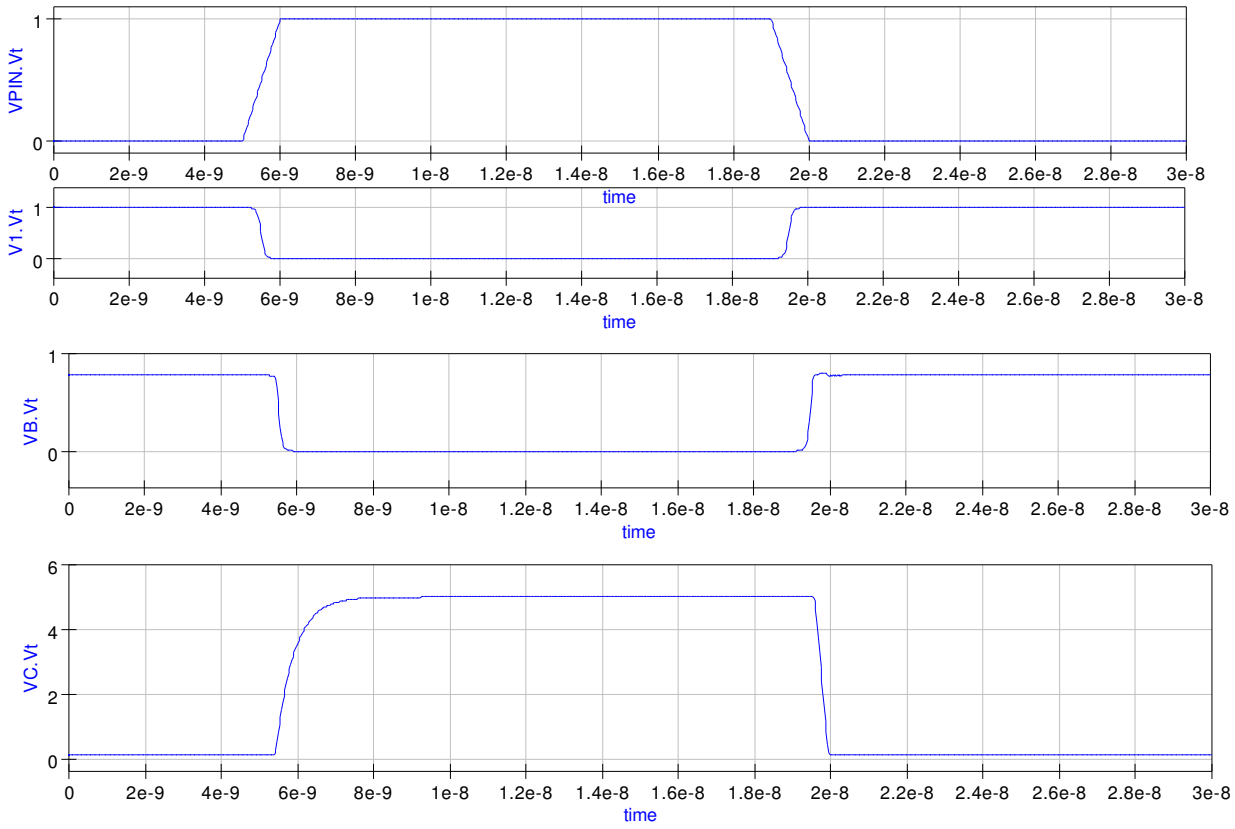


Figure 6.22: Digital TimeList waveforms for the circuit shown in Fig. 6.21

- Example 3: A more complex mixed-mode simulation example.

  The circuit shown in Fig. 6.23 brings together a number of the ideas outlined in these tutorial notes. A 4-bit digital signal is generated from a simple asynchronous binary counter operated from a digital clock signal. The counter output is transformed to the analogue domain using a simple node-bridge, of the type introduced in mixed-mode example 1. A 4-bit binary weighted DAC converts the transformed node-bridge signals into the final analogue output signal. The DAC operational amplifier is modelled as a gain block with a single pole frequency response and DC voltage output limiting. The output waveforms for this example are shown in Fig. 6.24 and the details of the operational amplifier model in Fig. 6.25.

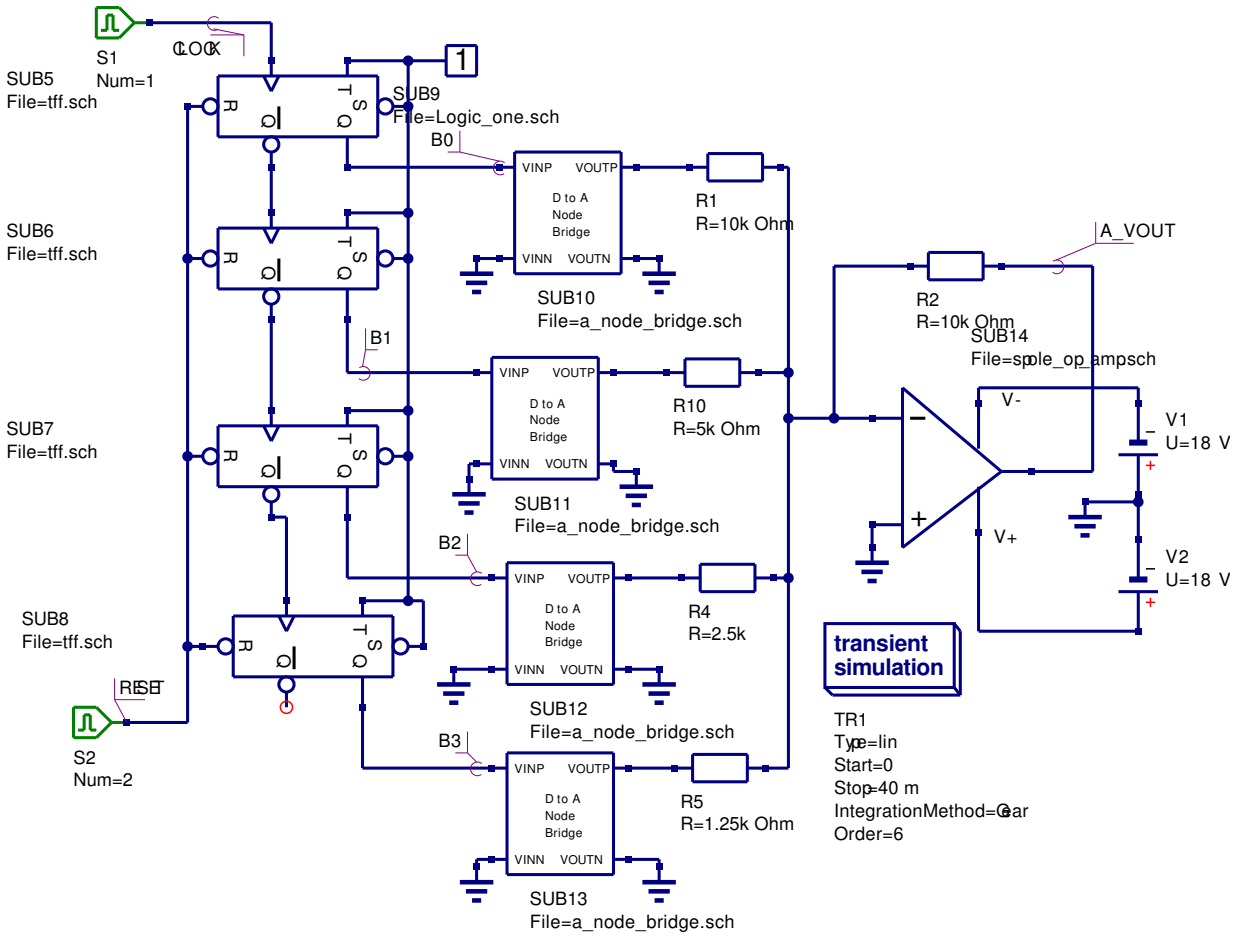


Figure 6.23: A more complex analogue-digital mixed-mode simulation example

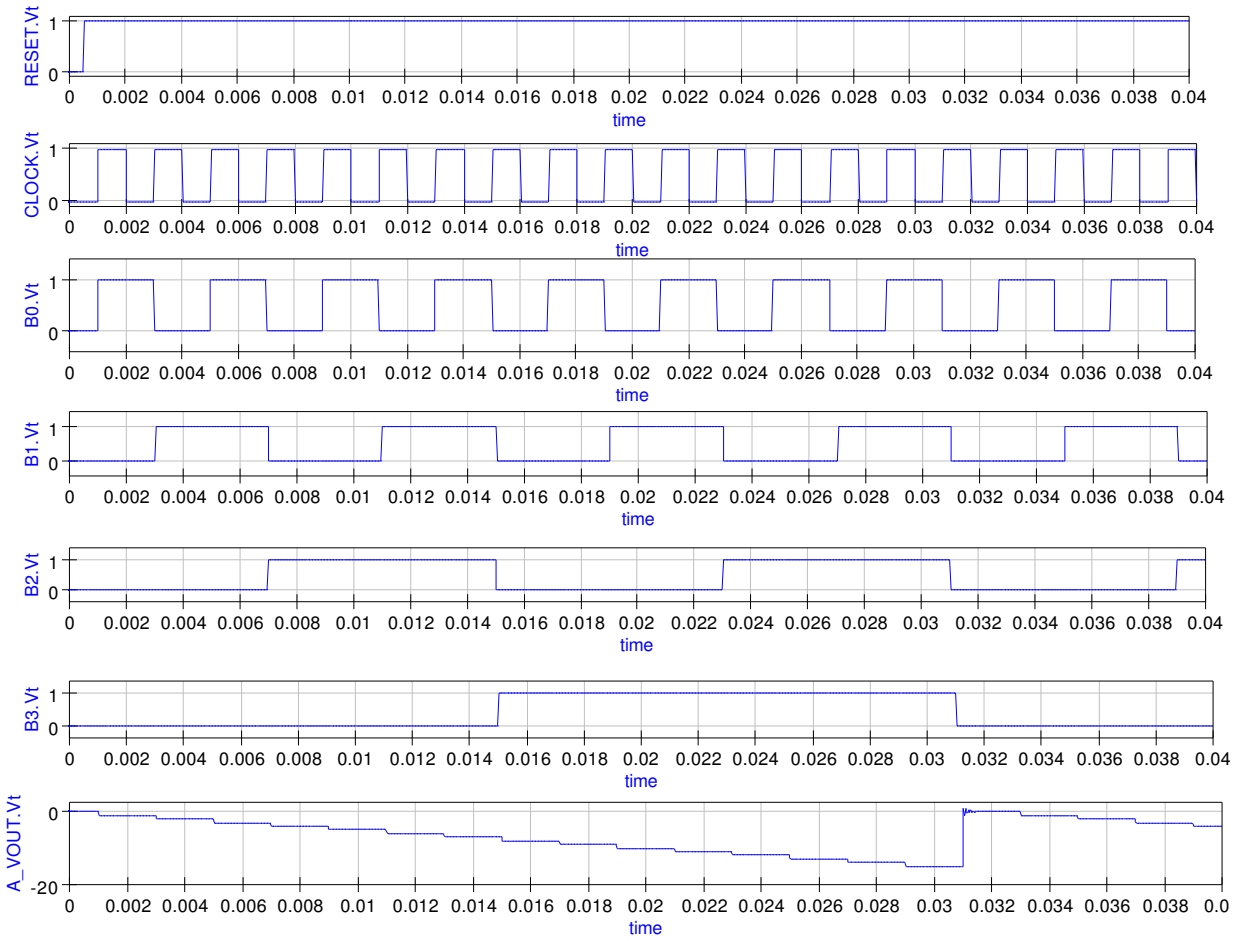


Figure 6.24: Digital TimeList waveforms for the circuit shown in Fig. 6.23

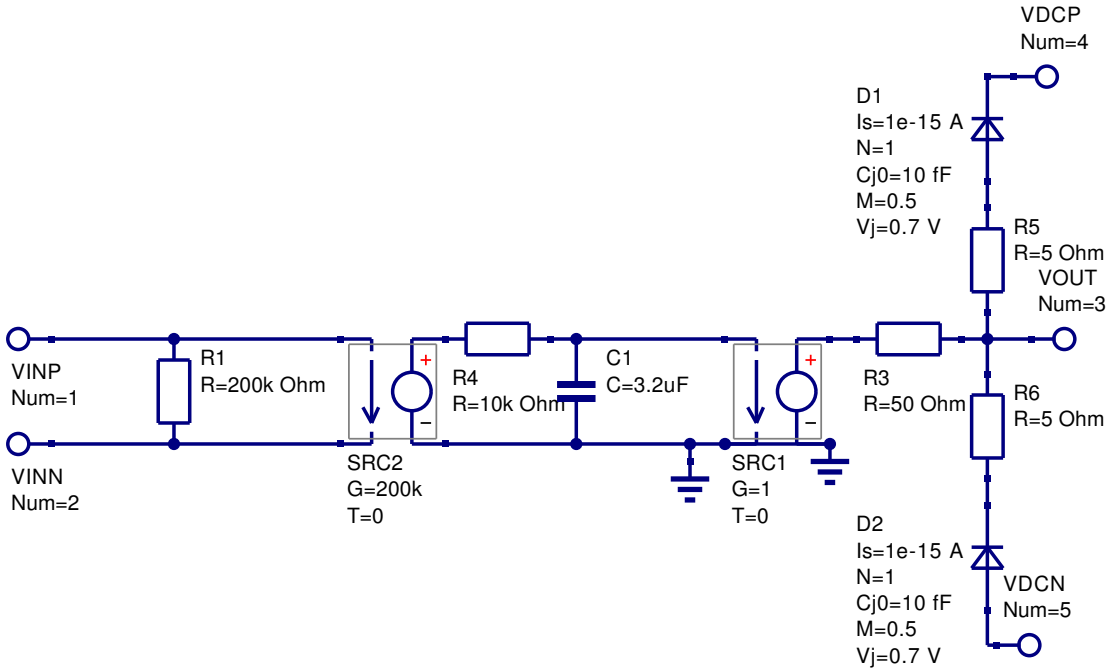


Figure 6.25: Operational amplifier model with Rin = 200k Ω, pole frequency = 5Hz, DC differential gain = 200k and Rout = 50 Ω

6.12 End Note

The examples described in these notes were all simulated using the latest CVS code version of Qucs. Since release of version 0.0.8, Qucs has matured enough to allow it to be used for mixed-mode simulation and many of the known bugs in Qucs 0.0.8 will be corrected with the release of Qucs 0.0.9 some time in the future. Release 0.0.9 will represent another important step in the development of a truly universal simulator. However, much more work needs to be done on the development of models for use across the different physical domains. My thanks to Michael Margraf and Stefan Jahn for all their hard work in correcting the bugs which surfaced while the examples presented in this tutorial note where being tested.

7 Modelling Operational Amplifiers

7.1 Introduction

Operation amplifiers (OP AMP) are a fundamental building block of linear electronics. They have been widely employed in linear circuit design since they were first introduced over thirty years ago. The use of operational amplifier models for circuit simulation using SPICE and other popular circuit simulators is widespread, and many manufacturers provide models for their devices. In most cases, these models do not attempt to simulate the internal circuitry at device level, but use macromodelling to represent amplifier behaviour as observed at the terminals of a device. The purpose of this tutorial note is to explain how macromodels can be used to simulate a range of the operational amplifier properties and to show how macromodel parameters can be obtained from manufacturers data sheets. This tutorial concentrates on models that can be simulated using Qucs release 0.0.9.

7.2 The Qucs built-in operational amplifier model

Qucs includes a model for an ideal operational amplifier. It's symbol can be found in the nonlinear components list. This model represents an operational amplifier as an ideal device with differential gain and output voltage limiting. The model is intended for use as a simple gain block and should not be used in circuit simulations where operational amplifier properties are crucial to overall circuit performance. Fig. 7.1 shows a basic inverting amplifier with a gain of ten, based on the Qucs OP AMP model. The simulated AC performance of this circuit is shown in Fig. 7.2. From Fig. 7.2 it is observed that the circuit gain and phase shift are constant and do not change as the frequency of the input signal is increased. This, of course, is an ideal situation which practical operational amplifiers do not reproduce. Let us compare the performance of the same circuit with the operational amplifier represented by a device level circuit. Shown in Fig. 7.3 is a transistor circuit diagram for the well known UA741 operational amplifier[1]. The gain and phase results for the circuit shown in Fig. 7.1, where the OP AMP is modelled by the UA741 transistor level model, are given in Fig. 7.4. The curves in this figure clearly illustrate the differences between the two simulation models. When simulating circuits that include operational

[1]The UA741 operational amplifier is one of the most studied devices. It is almost unique in that a transistor level model has been constructed for the device. Details of the circuit operation and modelling of this device can be found in (1) Paul R. Grey et. al., Analysis and Design of Analog Integrated Circuits, Fourth Edition, 2001, John Wiley and Sons INC., ISBN 0-471-32168-0, and (2) Andrei Vladimirescu, The SPICE book, 1994, John Wiley and Sons, ISBN 0-471-60926-9.

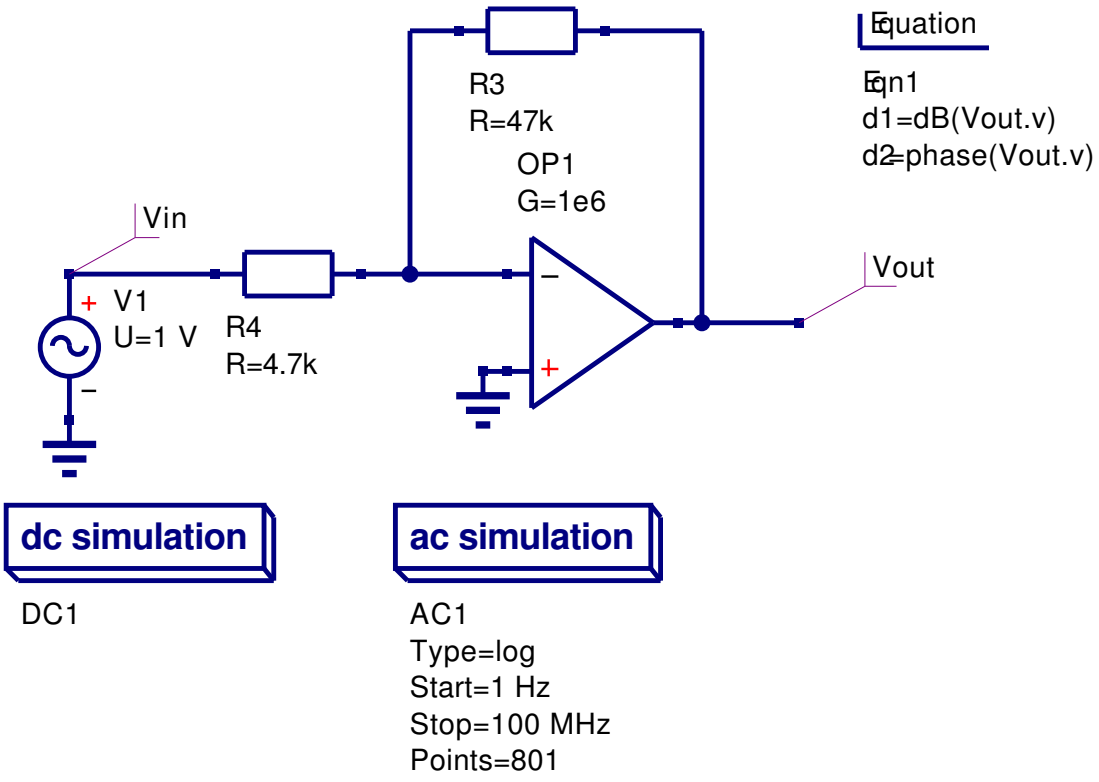


Figure 7.1: Qucs schematic for a basic OP AMP inverting amplifier:Qucs OP AMP has G=1e6 and Umax=15V.

amplifiers the quality of the OP AMP model can often be a limiting factor in the accuracy of the overall simulation results. Accurate OP AMP models normally include a range of the following device characteristics: (1) DC and AC differential gain, (2) input bias current, (3) input current and voltage offsets, (4) input impedance, (5) common mode effects, (6) slew rate effects, (7) output impedance, (8) power supply rejection effects, (9) noise, (10) output voltage limiting, (11) output current limiting and (12) signal overload recovery effects. The exact mix of selected properties largely depends on the purpose for which the model is being used; for example, if a model is only required for small signal AC transfer function simulation then including the output voltage limiting section of an OP AMP model is not necessary or indeed may be considered inappropriate. In the following sections of this tutorial article macromodels for a number of the OP AMP parameters listed above are developed and in each case the necessary techniques are outlined showing how to derive macromodel parameters from manufacturers data sheets.

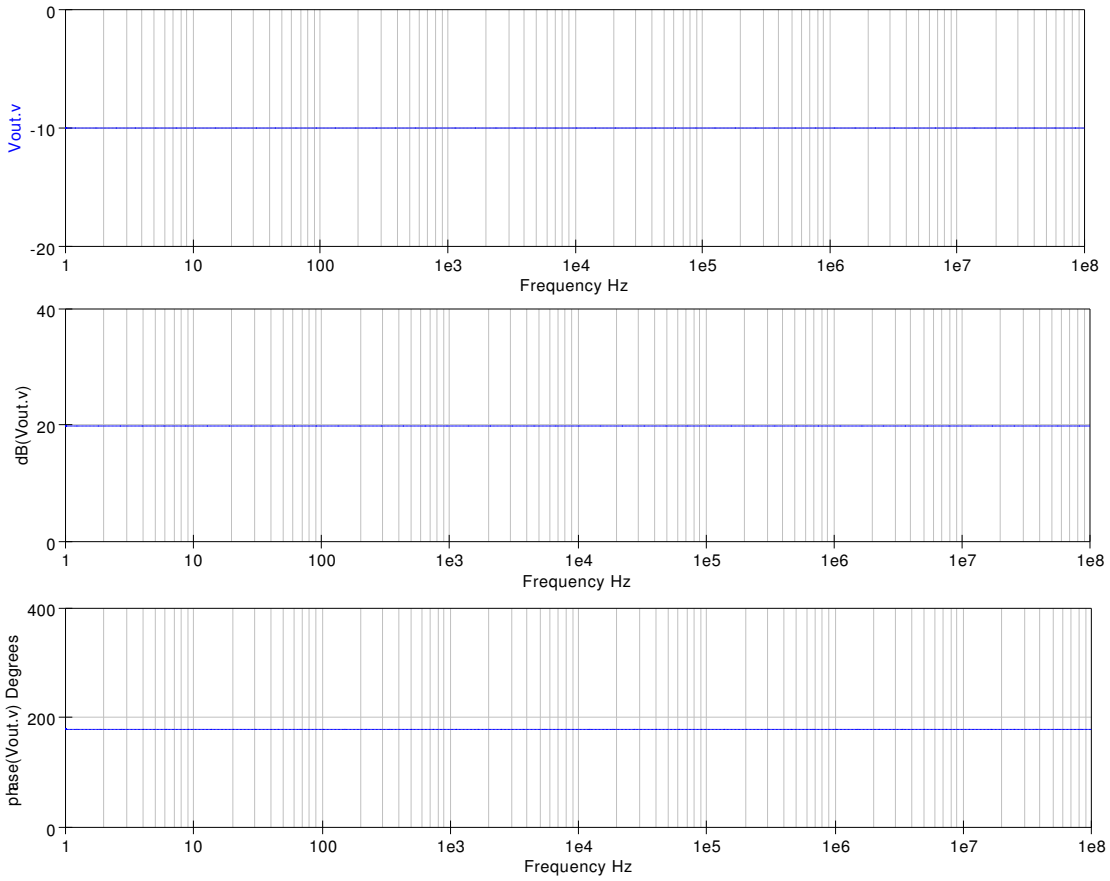


Figure 7.2: Gain and phase curves for a basic OP AMP inverting amplifier.

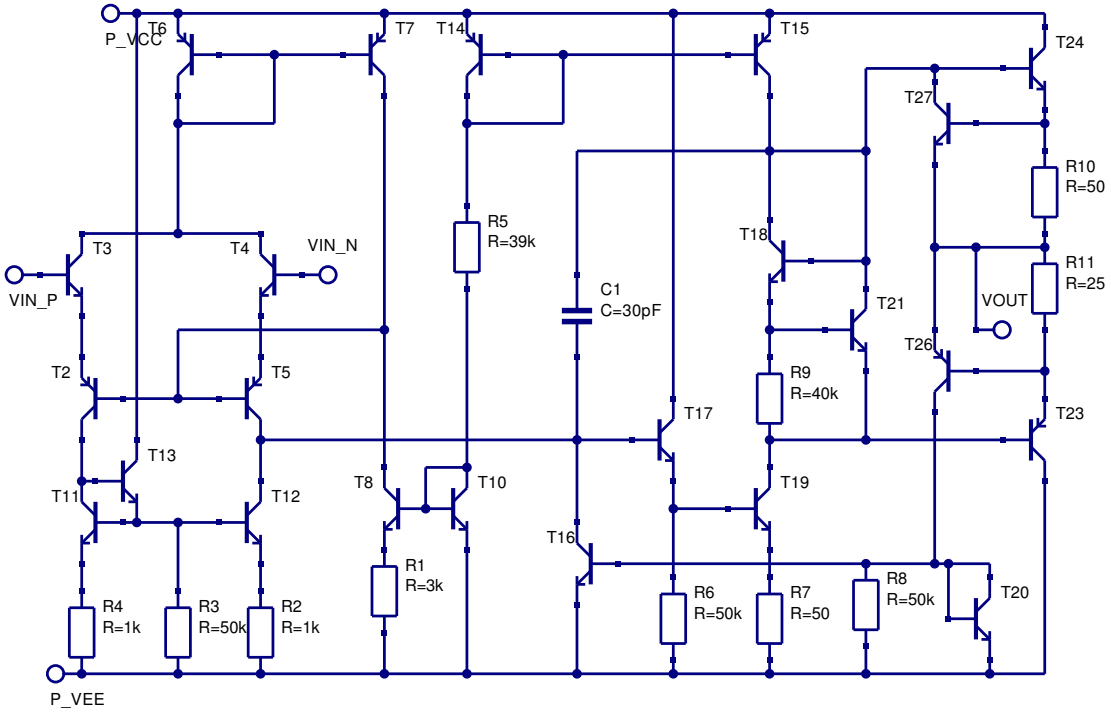


Figure 7.3: Transistor level circuit for the UA741 operational amplifier.

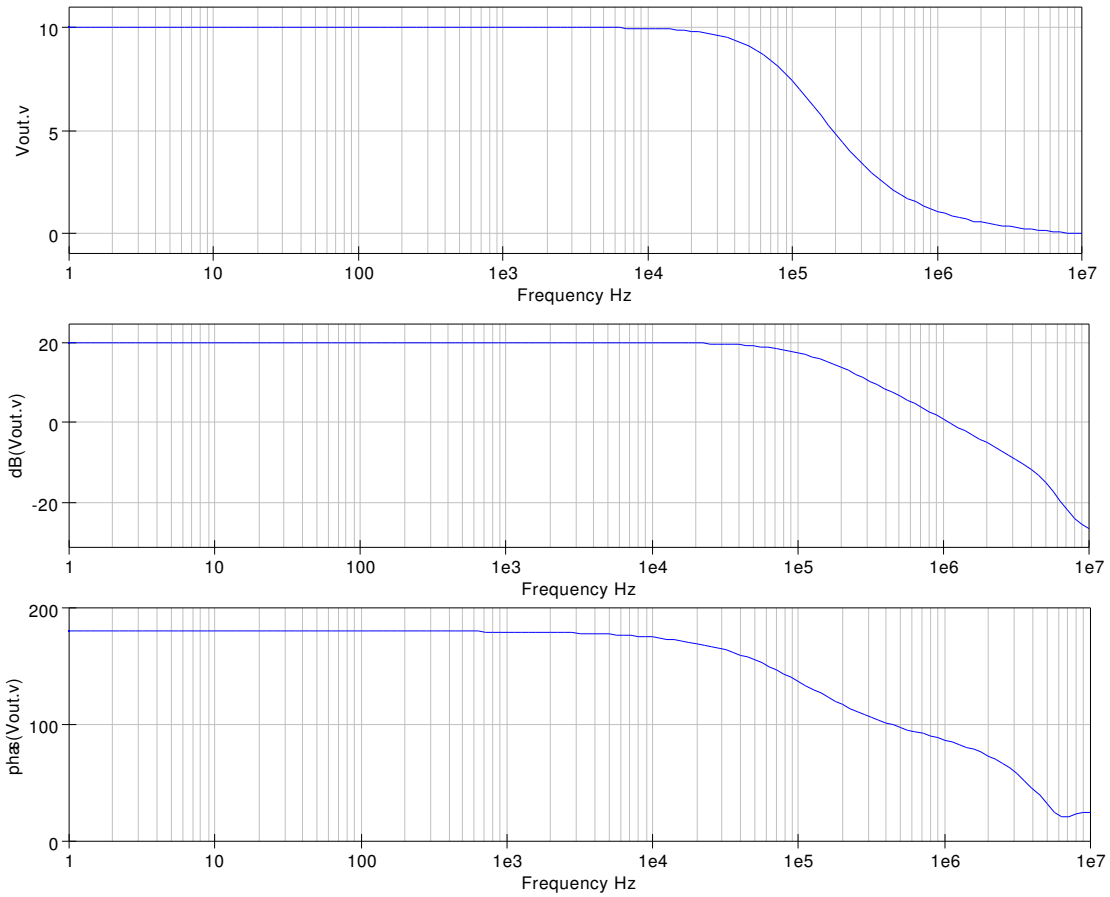


Figure 7.4: Gain and phase curves for a times 10 inverting amplifier with the OP AMP represented by a transistor level UA741 model.

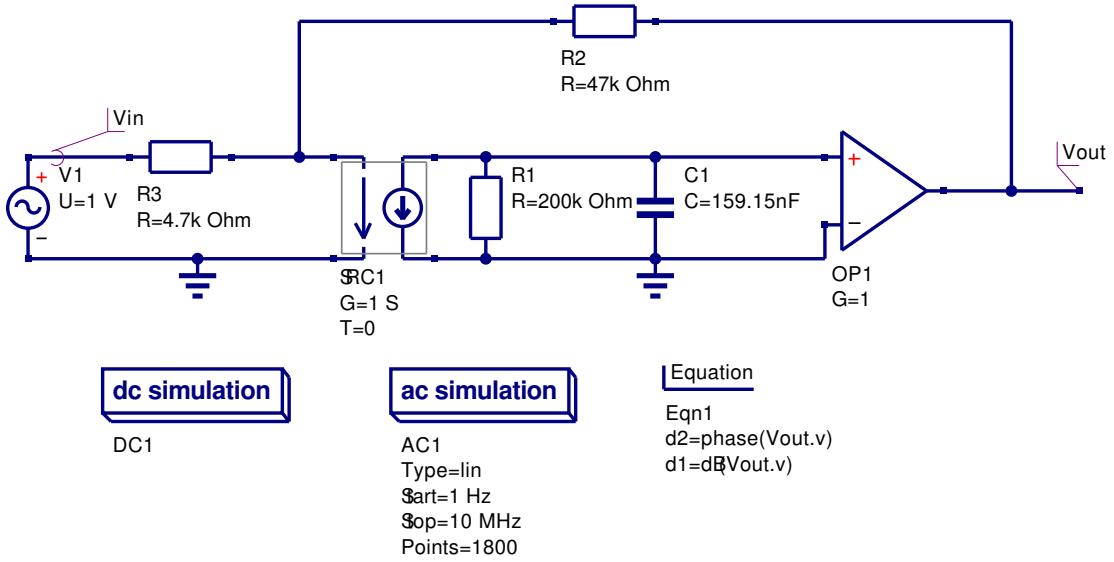


Figure 7.5: Modified Qucs OP AMP model to include single pole frequency response.

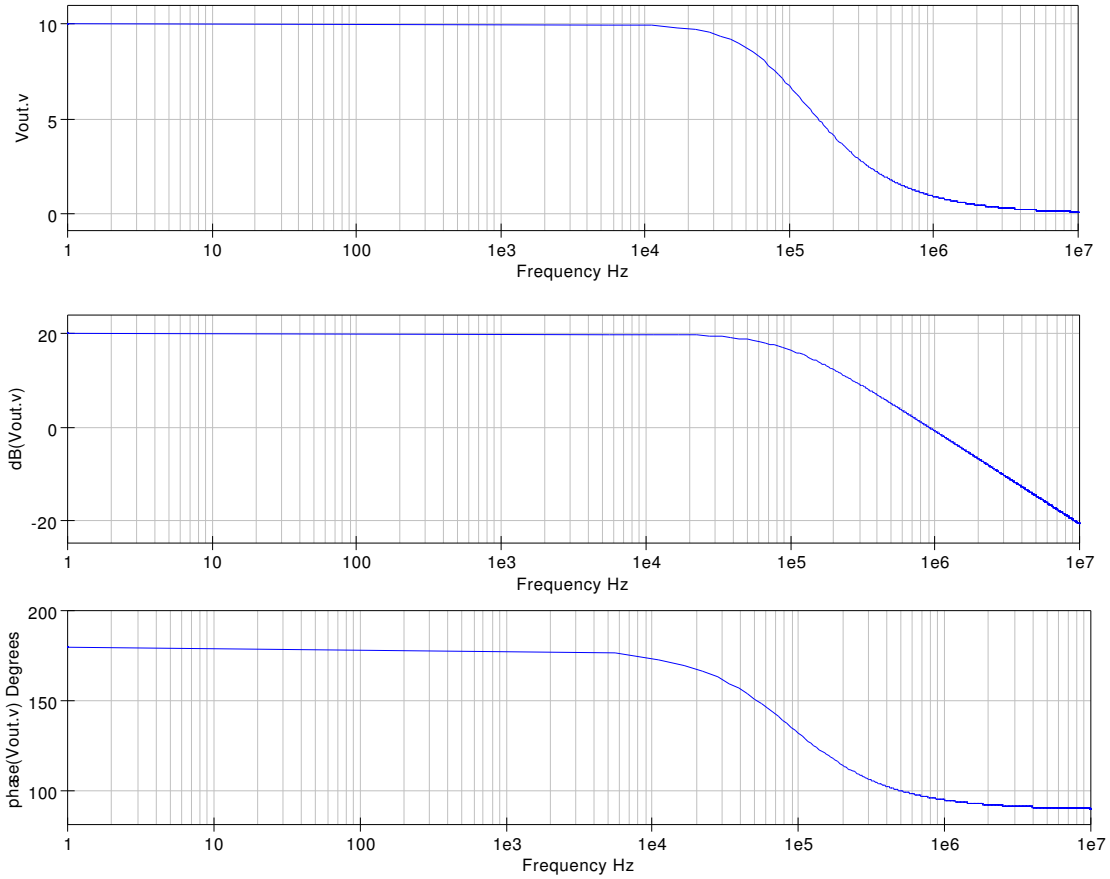


Figure 7.6: Gain and phase curves for the circuit shown in Fig. 7.5.

7.3 Adding features to the Qucs OP AMP model

In the previous section it was shown that the Qucs OP AMP model had a frequency response that is independent of frequency. By adding external components to the Qucs OP AMP model the functionality of the model can be improved. The UA741 differential open loop gain has a pole at roughly 5Hz and a frequency response that decreases at 20 dB per frequency decade from the first pole frequency up to a second pole frequency at roughly 3 MHz. The circuit shown in Fig. 7.5 models the differential frequency characteristics of a UA741 from DC to around 1 MHz. Figure 7.6 illustrates the closed loop frequency response for the modified Qucs OP AMP model.

7.4 Modular operational amplifier macromodels

Macromodelling is a term given to the process of modelling an electronic device as a "black box" where individual device characteristics are specified in terms of the signals, and other properties, observed at the input and output terminals of the black box. Such models operate at a functional level rather than at the more detailed transistor circuit level, offering considerable gain in computational efficiency.[2] Macromodels are normally derived directly from manufacturers data sheets. For the majority of operational amplifiers, transistor level models are not normally provided by manufacturers. One notable exception being the UA741 operational amplifier shown in Fig. 7.3.
A block diagram of a modular[3] general purpose OP AMP macromodel is illustrated in Fig. 7.7. In this diagram the blocks represent specific amplifier characteristics modelled by electrical networks composed of components found in all the popular circuit simulators[4]. Each block consists of one or more components which model a single amplifier parameter or a group of related parameters such as the input offset current and voltage. This ensures that changes to one particular parameter do not indirectly change other parameters. Local nodes and scaling are also employed in the macromodel blocks. Furthermore, because each block operates separately, scaled voltages do not propagate outside individual blocks. Each block can be modelled with a Qucs subcircuit that has the required specification and buffering from other blocks. Moreover, all subcircuits are self contained entities where the internal circuit details are hidden from other blocks. Such an approach is similar to structured high-level computer programming where the internal details of functions are hidden from

[2] Computational efficiency is increased mainly due to the fact that operational amplifier macromodels have, on average, about one sixth of the number of nodes and branches when compared to a transistor level model. Furthermore, the number of non-linear p-n junctions included in a macromodel is often less than ten which compares favorable with the forty to fifty needed to model an amplifier at transistor level.

[3] Brinson M. E. and Faulkner D. J., Modular SPICE macromodel for operational amplifiers, IEE Proc.-Circuits Devices Syst., Vol. 141, No. 5, October 1994, pp. 417-420.

[4] Models employing non-linear controlled sources, for example the SPICE B voltage and current sources, are not allowed in Qucs release 0.0.9. Non-linear controlled sources are one of the features on the Qucs to-do list.

users. Since the device characteristics specified by each block are separate from all other device characteristics only those amplifier characteristics which are needed are included in a given macromodel. This approach leads to a genuinely structured macromodel. The following sections present the detail and derivation of the electrical networks forming the blocks drawn in Fig. 7.7. To illustrate the operation of the modular OP AMP macromodel the values of the block parameters are calculated for the UA741 OP AMP and used in a series of example simulations. Towards the end of this tutorial note data are presented for a number of other popular general purpose operational amplifiers.

7.5 A basic AC OP AMP macromodel.

A minimum set of blocks is required for the modular macromodel to function as an amplifier: an input stage, a gain stage and an output stage. These form the core modules of all macromodels.

7.5.1 The input stage.

The input stage includes amplifier offset voltage, bias and offset currents, and the differential input impedance components. The circuit for the input stage is shown in Fig. 7.8, where

1. *R1* = *R2* = Half of the amplifier differential input resistance (*RD*).
2. *Cin* = The amplifier differential input capacitance (*CD*).
3. *Ib1* = *Ib2* = The amplifier input bias current (*IB*).
4. *Ioff* = Half the amplifier input offset current (*IOFF*).
5. *Voff1* = *Voff2* = Half the input offset voltage (*VOFF*).

Typical values for the UA741 OP AMP are:

1. *RD* = 2 MΩ and *R1* = *R2* = 1MΩ
2. *CD* = *Cin1* = 1.4 pF.
3. *IB* = *Ib1* = *Ib2* = 80 nA.
4. *IOFF* = 20 nA and *Ioff1* = 10 nA.
5. *VOFF* = 0.7 mV and *Voff1* = *Voff2* = 0.35 mV.

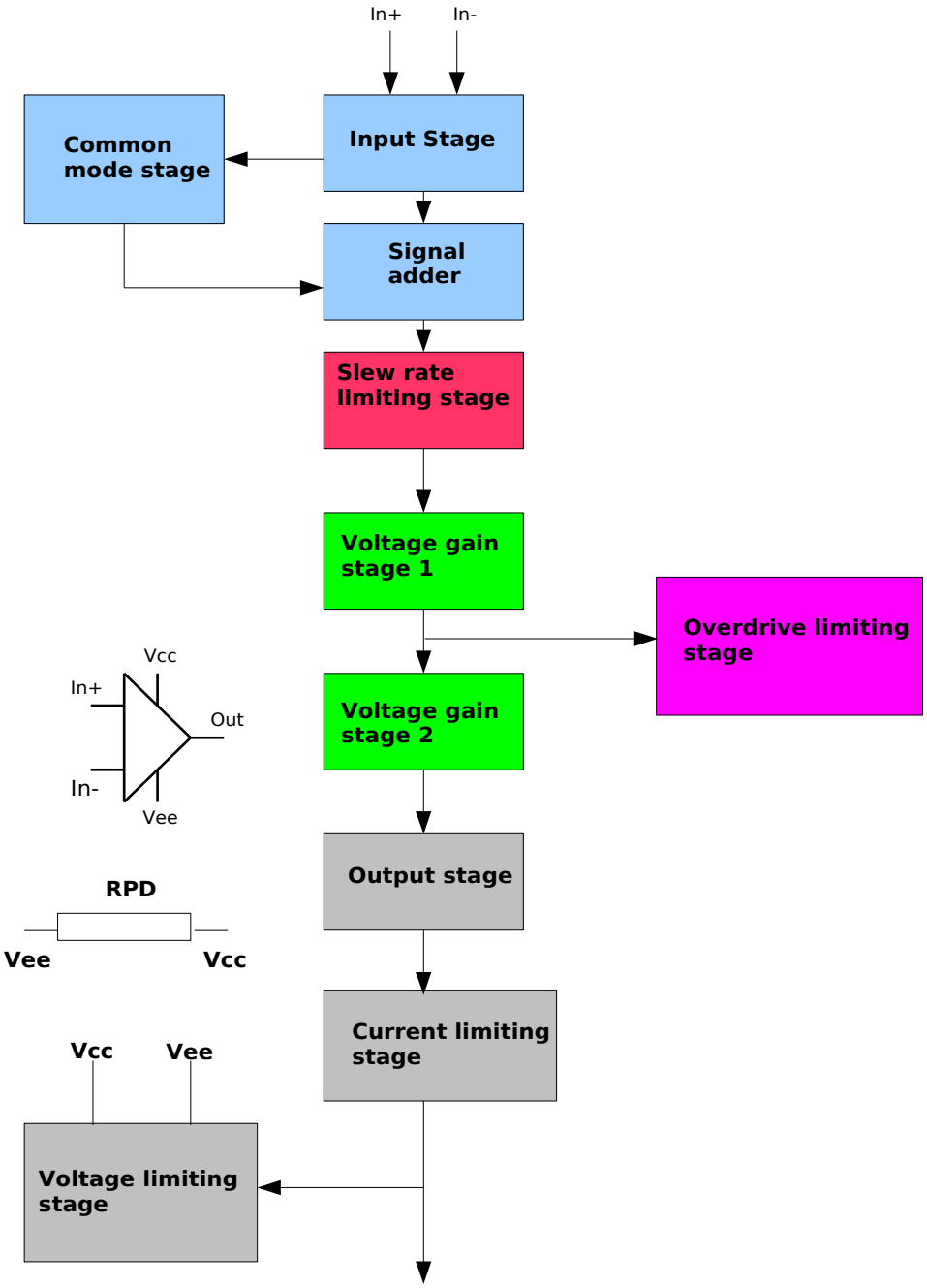


Figure 7.7: Block diagram of an operational amplifier macromodel.

The differential output signal (VD) is given by $VD\_P1 - VD\_N1$ and the common mode output signal (VCM) by $(VD\_P1 + VD\_N1)/2$.

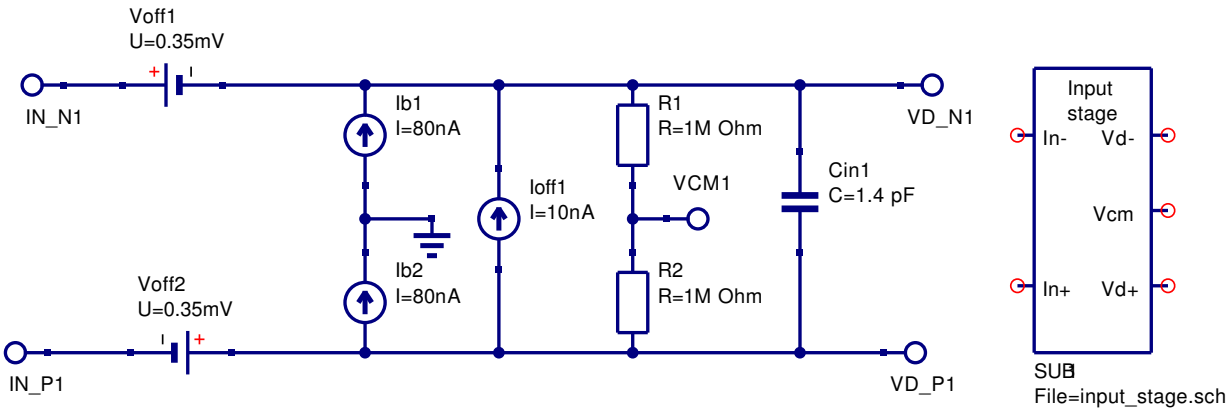


Figure 7.8: Modular OP AMP input stage block.

7.5.2 Voltage gain stage 1.

The circuit for voltage gain stage 1 is shown in Fig. 7.9, where

1. $RD1$ = 100 MΩ = A dummy input resistor - added to ensure nodes IN_P1 and IN_N1 are connected by a DC path.
2. $GMP1$ = 1 S = Unity gain voltage controlled current generator.
3. $RADO$ = The DC open loop differential gain ($AOL(DC)$) of the OP AMP.
4. $CP1$ = $1/(2*\pi*GBP)$, where GBP = the OP AMP gain bandwidth product.

Typical values for the UA741 OP AMP are:

1. $RADO$ = 200kΩ. ($AOL(DC)$ = 106 dB)
2. $CP1$ = 159.15 nF (The typical value for UA741 GBP is 1 MHz).

7.5.3 Derivation of voltage gain stage 1 transfer function

Most general purpose operational amplifiers have an open loop differential voltage gain which has (1) a very high value at DC (2) a dominant pole ($fp1$) at a low frequency - typically below 100 Hz, and (3) a gain response characteristic that rolls-off at 20 dB per decade up to a unity gain frequency which is often in the MHz region. This form of response has a constant gain bandwidth product (GBP) over the frequency range from $fp1$ to GBP. A typical OP AMP differential open loop response is shown in Fig. 7.10. The voltage gain transfer function for this type of characteristic can be modelled with the electrical network given in Fig. 7.9, where the the AC voltage transfer function is

$$vout(POLE\_1\_OUT1) = \frac{GMP1 * (V(IN\_P1) - V(IN\_N1)) * RADO}{1 + j(\omega * RADO * CP1)} \tag{7.1}$$

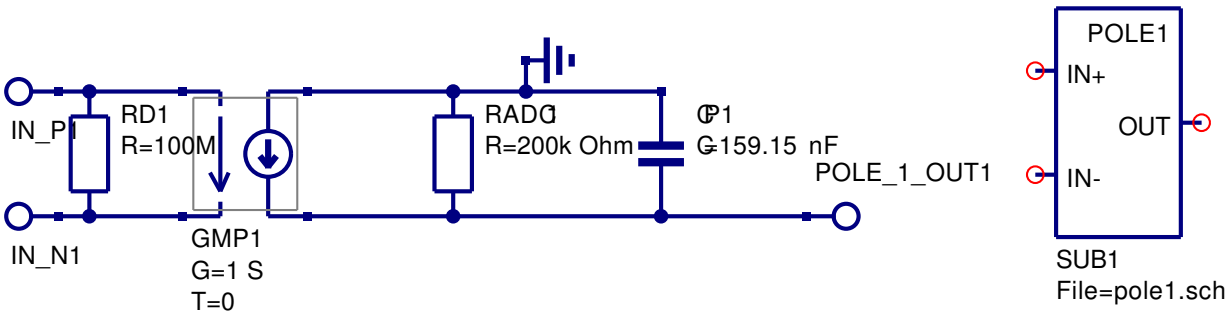


Figure 7.9: Modular OP AMP first voltage gain stage.

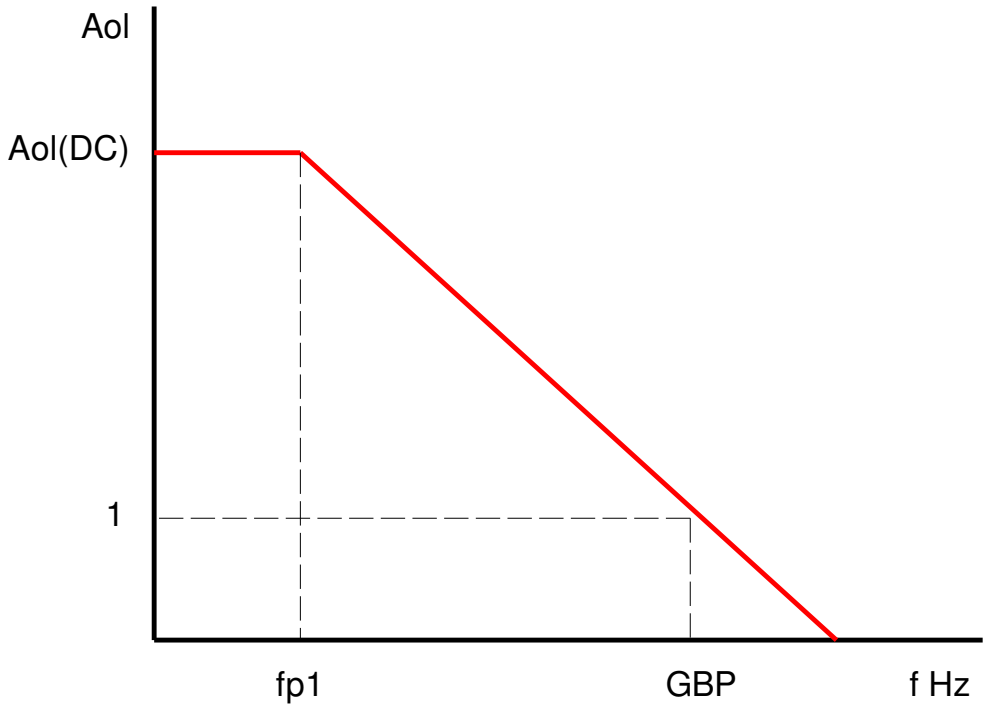


Figure 7.10: OP AMP open loop differential voltage gain as a function of frequency.

Where

$$f_{P1} = \frac{1}{2\pi * RADO * CP1} \tag{7.2}$$

Let $RADC = Aol(DC)$ and $GMP1 = 1\ S$. Then, because $fp1{*}AOL(DC) = GBP$,

$$CP1 = \frac{1}{2\pi * GBP} \tag{7.3}$$

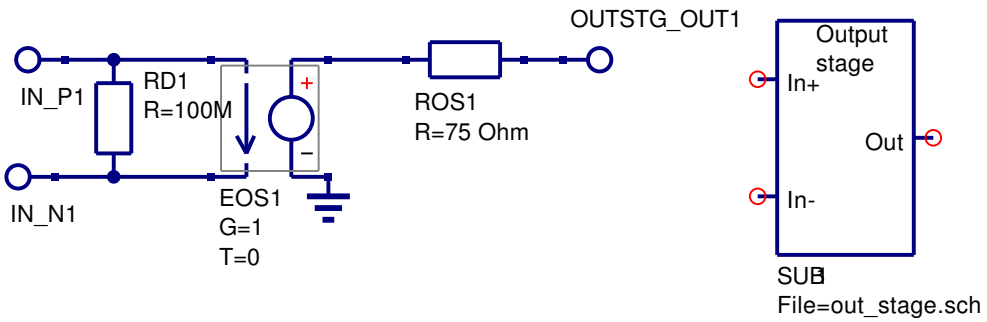


Figure 7.11: Modular macromodel output stage.

7.5.4 Output stage.

The electrical network representing a basic output stage is given in Fig. 7.11, where

1. $RD1$ = 100 MΩ = A dummy input resistor - added to ensure nodes IN_P1 and IN_N1 are connected by a DC path.
2. $EOS1$ G = 1 = Unity gain voltage controlled voltage generator.
3. $ROS1$ = OP AMP output resistance.

A typical value for the UA741 OP AMP output resistance is $ROS1 = 75\Omega$.

7.5.5 A subcircuit model for the basic AC OP AMP macromodel

The model for the basic AC OP AMP macromodel is shown in Fig. 7.12. The input stage common mode voltage (Vcm) is not used in this macromodel and has been left floating. To test the performance of the AC macromodel it's operation was compared to the transistor level UA741 model. Figure 7.13 shows a schematic circuit for two inverting amplifiers, each with a gain of ten, driven from a common AC source. One of the amplifiers uses the simple AC macromodel and the other the transistor level UA741 model. Figure 7.14 illustrates the output gain and phase curves for both amplifiers. In general the plotted curves are very similar. However, at frequencies above the GBP frequency the basic AC macromodel does not correctly model actual OP AMP performance. This is to be expected because the simple AC macromodel does not include any high frequency modelling components. Notice also that the DC output voltages for vout and vout3 are very similar, see the DC tabular results given in Fig. 7.13.

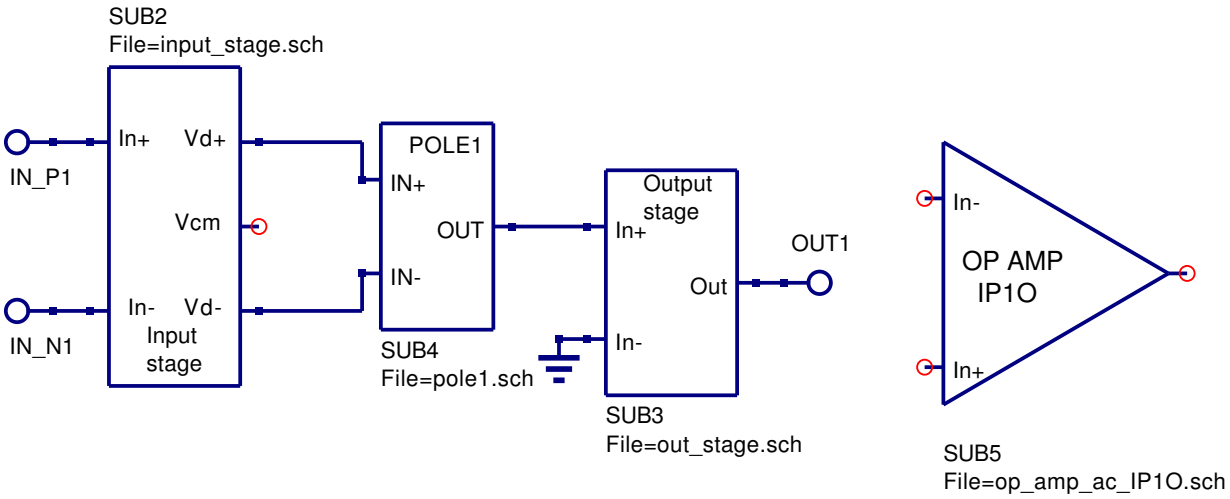


Figure 7.12: Simple AC OP AMP macromodel.

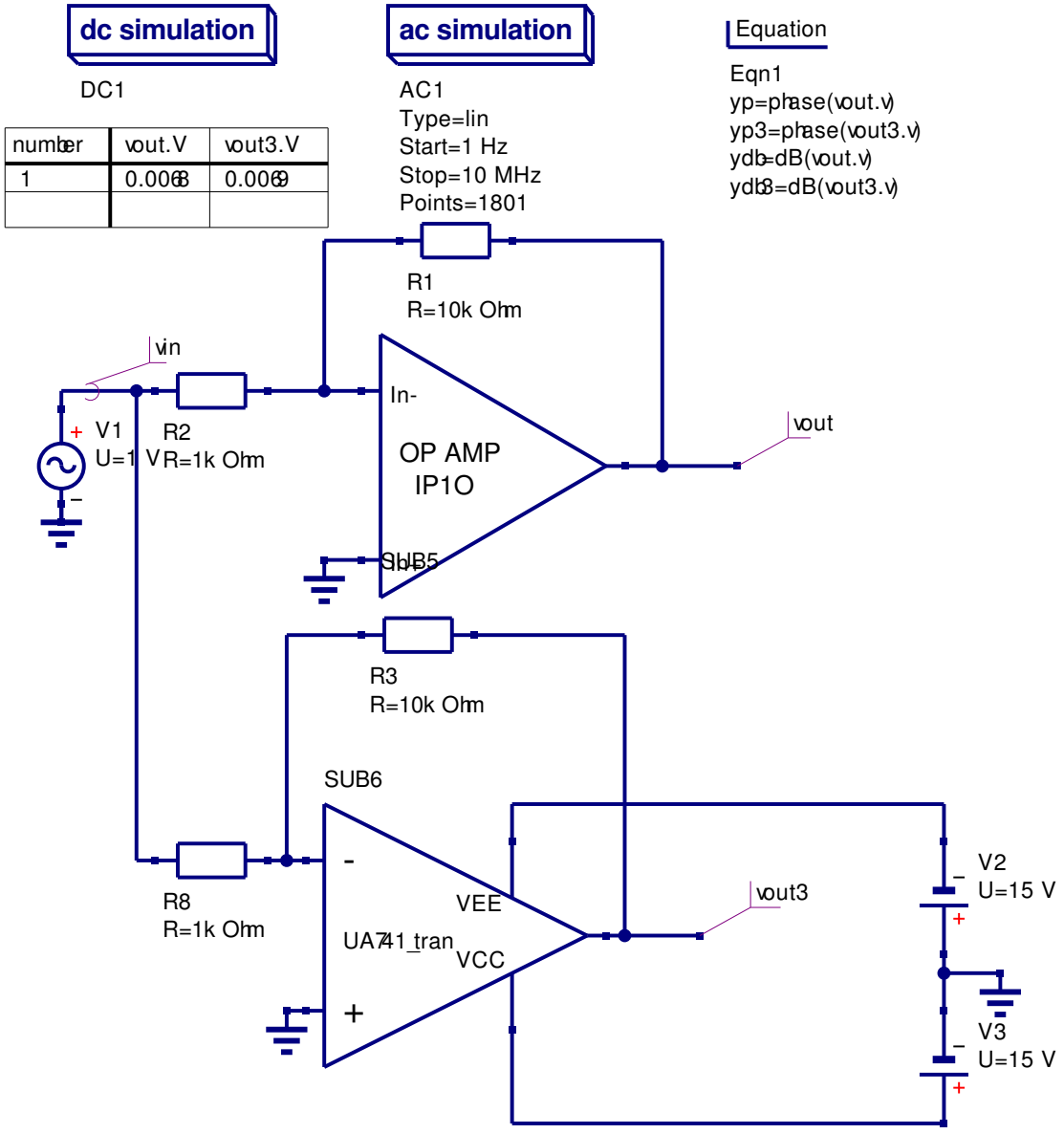


Figure 7.13: Test circuit for an inverting amplifier. Output signals: (1) vout for AC macro-model, (2) vout3 for UA741 transistor model.

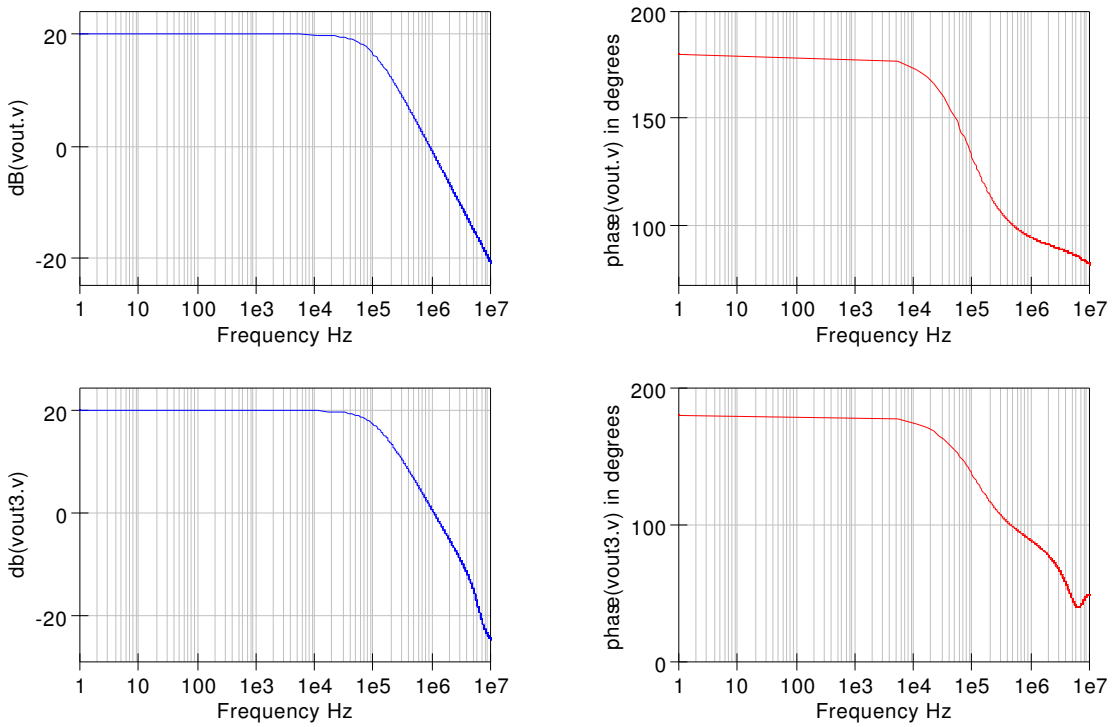


Figure 7.14: Simulation test results for the circuit shown in Fig. 7.13.

7.6 A more accurate OP AMP AC macromodel

Most general purpose OP AMPs have a high frequency pole in their differential open loop gain characteristics. By adding a second gain stage to the simple AC macromodel the discrepancy in the high frequency response can be corrected. The model for the second gain stage is shown in Fig. 7.15. This additional gain stage has a structure similar to the first gain stage, where

1. *RD2* = 100 MΩ = A dummy input resistor - added to ensure nodes `IN_P2` and `IN_N2` are connected by a DC path.
2. *GMP2* = 1 S = Unity gain voltage controlled current generator.
3. *RP2* = 1Ω.
4. *CP2 = 1/(2π *fp2)*, where $fp2$ = the second pole frequency in Hz.

A typical value for the UA741 OP AMP high frequency pole is *fp2* = 3M Hz

7.6.1 Derivation of voltage gain stage 2 transfer function.

The differential voltage gain transfer function for voltage gain stage 2 is given by

$$vout(POLE\_2\_OUT1) = \frac{GMP2 * (V(IN\_P2) - V(IN\_N2)) * RP2}{1 + j(\omega * RP2 * CP2)} \tag{7.4}$$

Let *RP2 = 1*Ω and *GMP2 = 1 S*. Then

$$vout(POLE\_2\_OUT1) = \frac{V(IN\_P2) - V(IN\_N2)}{1 + j(\omega * CP2)} \tag{7.5}$$

and

$$CP2 = \frac{1}{2\pi * fp2} \tag{7.6}$$

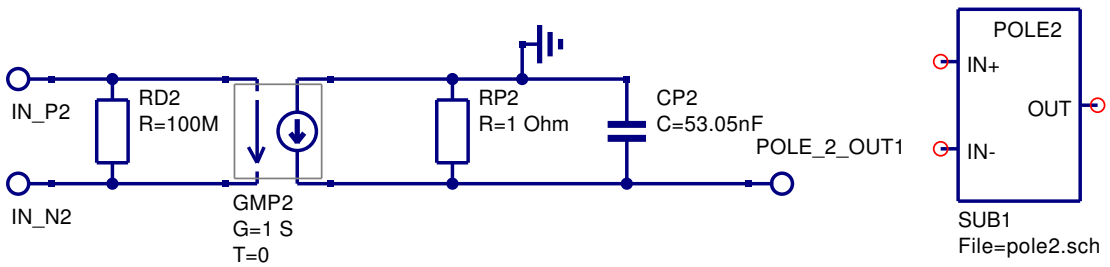


Figure 7.15: Modular OP AMP second voltage gain stage.

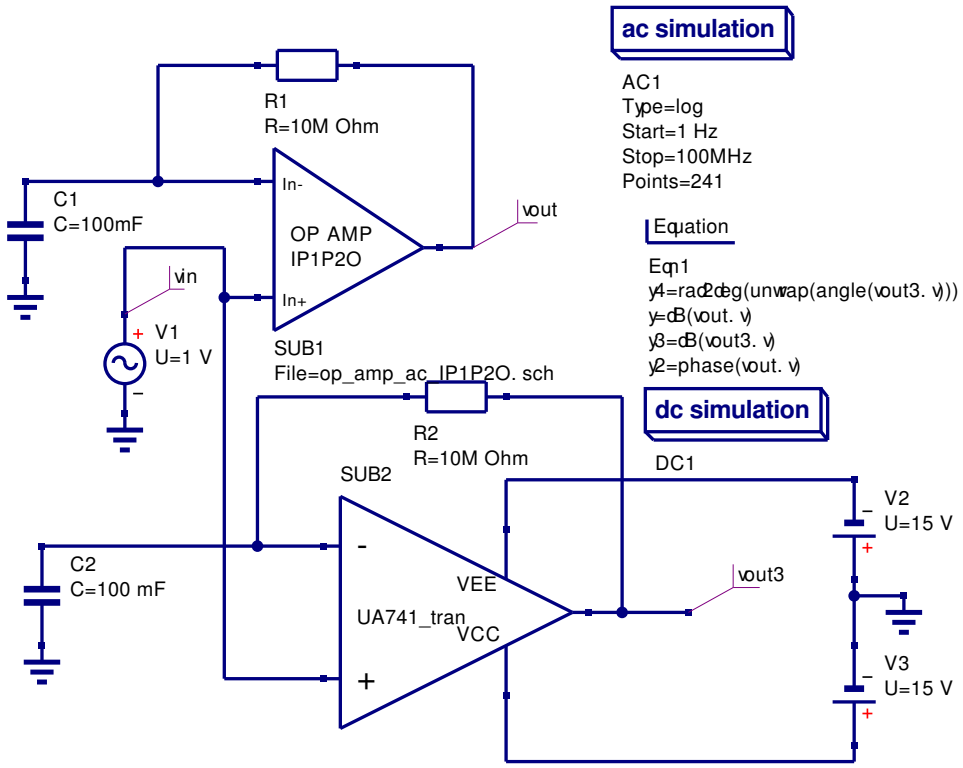


Figure 7.16: Test circuit for simulating OP AMP open loop differential gain.

7.6.2 Simulating OP AMP open loop differential gain

The circuit shown in Fig. 7.16 allows the open loop differential gain *(Aol(f))* to be simulated. This circuit employes a feedback resistor to ensure DC stability. Fig. 7.16 illustrates two test circuits driven from a common AC source. This allows the performance of the AC macromodel and the UA741 transistor level model to be compared. The AC voltage transfer function for the test circuit is

$$vout(f) = \frac{Aol(f)}{1 + \dfrac{Aol(f)}{1 + j\omega * R * C}} vin(f) \tag{7.7}$$

where $vout(f) = (V^+ - V^-) * Aol(f)$, $V^+ = vin(f)$, and $V^- = \dfrac{vout(f)}{1 + j\omega * R * C}$

Provided $\dfrac{Aol(f)}{\omega * R * C} << 1$, equation (7) becomes $vout(f) \Rightarrow Aol(f) * vin(f)$. Hence, for those frequencies where this condition applies *vout(f) = Aol(f)* when *vin(f)* = 1 V. Figure 17 shows plots of the open loop simulation data. Clearly with the test circuit time constant set at 1e6 seconds the data is accurate for frequencies down to 1 Hz.

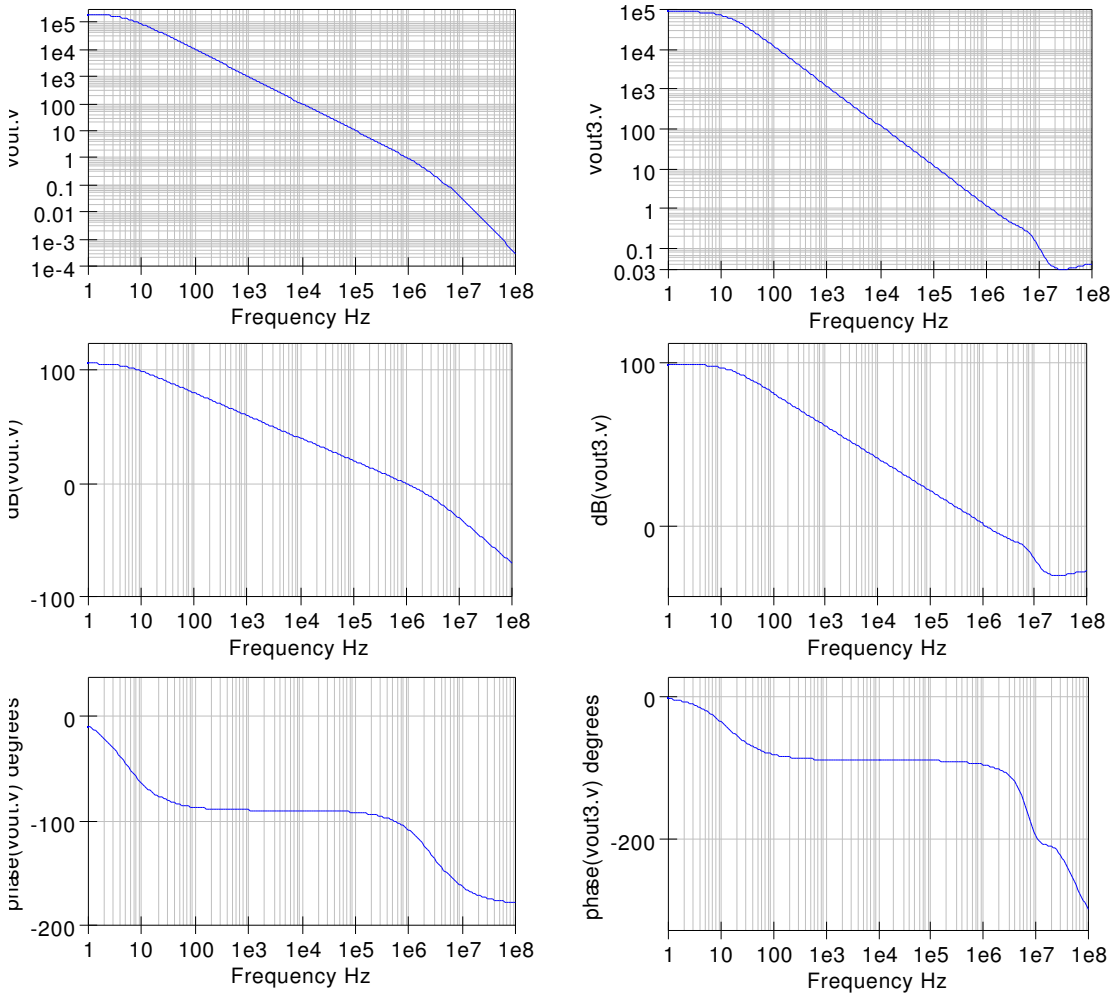


Figure 7.17: Simulation test results for the circuit shown in Fig. 7.16.

7.7 Adding common mode effects to the OP AMP AC macromodel

The open-loop differential gain $A_D(f)$ for most general purpose operational amplifiers can be approximated by

$$A_D(f) = AD(0)\frac{1}{1+j\frac{f}{f_{PD}}} \tag{7.8}$$

Similarly, the common-mode gain $A_{CM}(f)$ can be represented by the same single-pole response and a single zero response given by

$$A_{CM}(f) = A_{CM}(0)\frac{1+j\frac{f}{f_{CMZ}}}{1+j\frac{f}{f_{PD}}} \tag{7.9}$$

Defining the common-mode rejection ratio $CMRR(f)$ of an OP AMP as

$$CMRR(f) = \frac{A_D(f)}{A_{CM}(f)} \tag{7.10}$$

gives

$$CMRR(f) = CMRR(0)\frac{1}{1+j\frac{f}{f_{CMZ}}} \tag{7.11}$$

where

$$CMRR(0) = \frac{A_D(0)}{A_{CM}(0)} \tag{7.12}$$

Common-mode effects can be added to OP AMP macromodels by including a stage in the modular macromodel that introduces a zero in the amplifier frequency response. Output V_{CM} from the macromodel input stage senses an amplifier common mode signal. This signal, when passed through a CR network generates the required common mode zero. Figure 18 gives the model of the zero generating network, where.

1. $RDCMZ = 650$ MΩ = common-mode input resistance/2.
2. $RCM1 = 1$ MΩ
3. $ECM1$ G $= 31.623 = \dfrac{\frac{RCM1}{RCM2}}{CMRR(0)}$. (NOTE: $RCM1/RCM2$is a scaling factor.)
4. $CCM1 = 795.8$ pF $= \dfrac{1}{2\pi * RCM1 * f_{CMZ}}$.

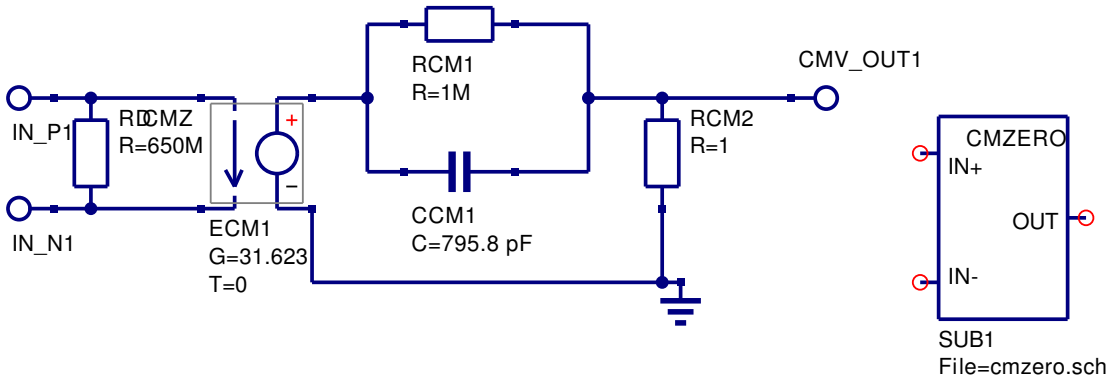


Figure 7.18: Common-mode zero macromodel

5. $RCM2 = 1\ \Omega$

Typical values for the UA741 OP AMP are:

1. Common-mode input resistance = 1300 MΩ.
2. $CMRR(0)$ = 90 dB
3. f_{CMZ} = 200 Hz.

The AC voltage transfer function for the common-mode zero transfer function is

$$Vout(\texttt{CMV\_OUT1}) = G(ECM1)\frac{RCM2}{RCM1}\left[\frac{1+j\omega * RCM1 * CCM1}{1+j\omega * RCM2 * CCM1}\right][V(\texttt{IN\_P1}) - V(\texttt{IN\_N1})] \tag{7.13}$$

As $\frac{RCM2}{RCM1} << 1$, the pole introduced by the common-mode RC network is at a very high frequency and can be neglected. Combining the common-mode zero with the previously defined stage models yields the macromodel shown in Fig. 7.19. In this model the differential and common-mode signals are combined using a simple analogue adder based on voltage conrolled current generators.

7.7.1 Simulating OP AMP common-mode effects

OP AMP common-mode effects can be simulated using the circuit shown in Fig. 7.20.[5] The resulting output voltages (vout.v and vout3.v) for a test circuit with matched resistors are shown plotted in Fig. 7.21, where $\frac{vout(0)}{vin} = \frac{1}{CMRR(0)}$. Clearly the test results

[5]Brinson M.E. and Faulkner D.J., New approaches to measurement of operational amplifier common-mode rejection ratio in the frequency domain, IEE Proc-Circuits Devices Sys., Vol 142, NO. 4, August 1995, pp 247-253.

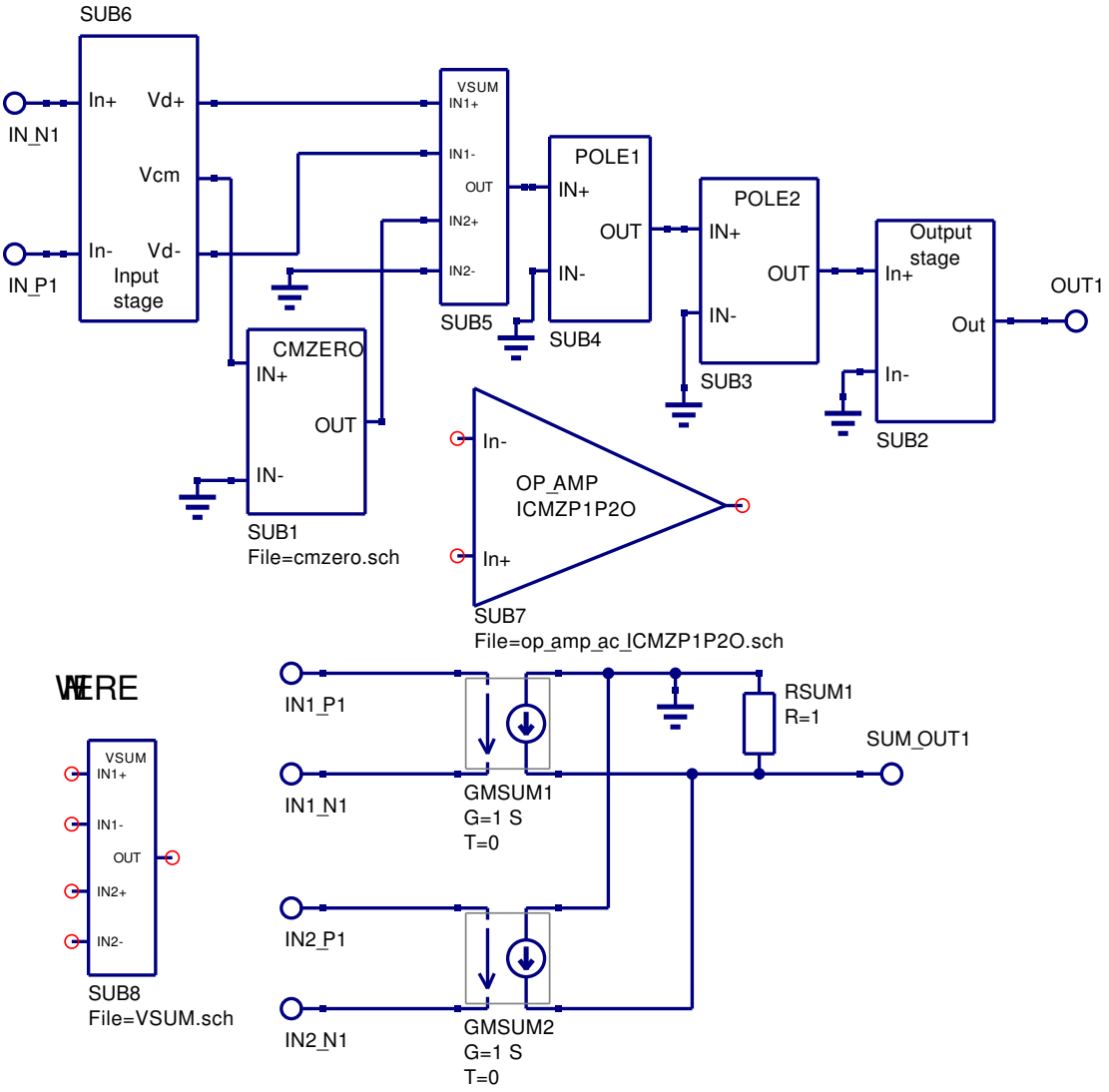


Figure 7.19: AC macromodel including common-mode zero.

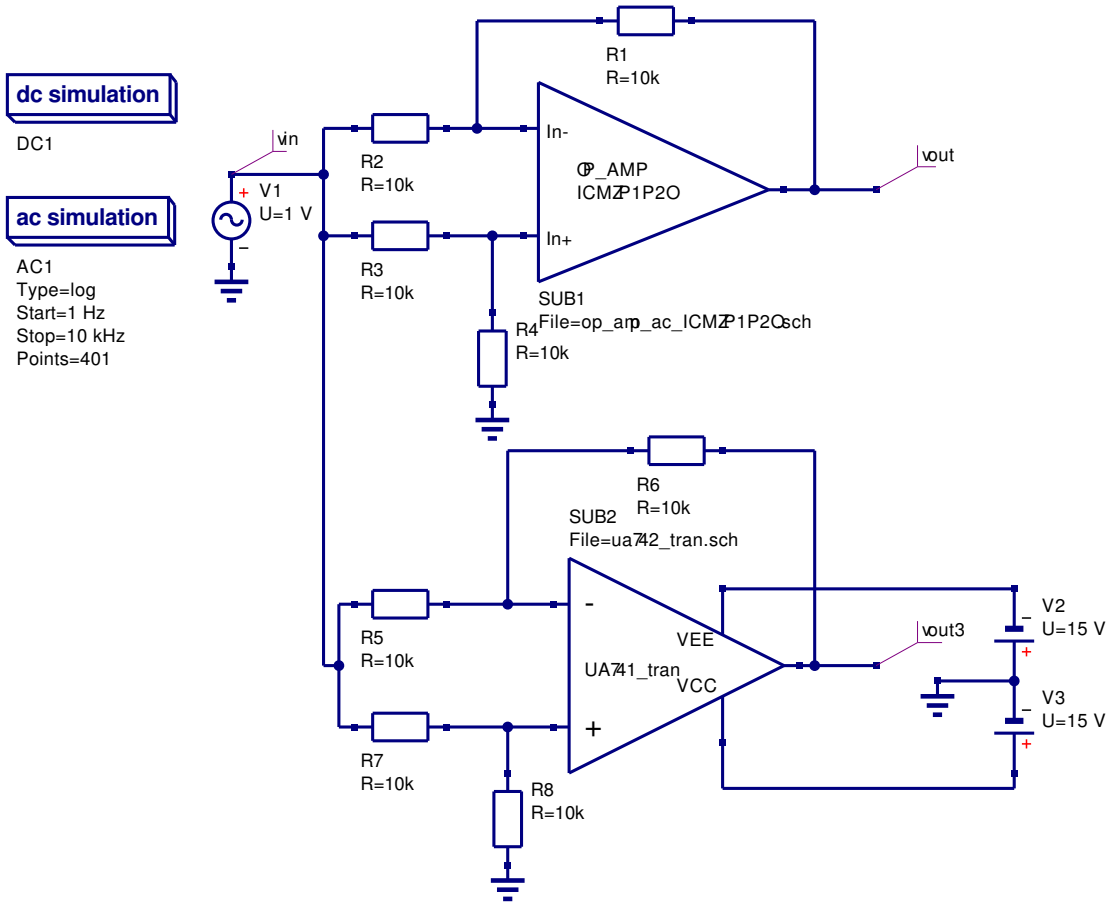


Figure 7.20: Simulation of OP AMP common-mode performance.

for the macromodel and the UA741 transistor model are very similar. In the case of the macromodel typical device parameters were used to calculate the macromodel component values. However, in the transistor level model the exact values of the component parameters are unknown.[6]

[6]The UA741 transistor level model is based on an estimate of the process parameters that determine the UA741 transistor characteristics. Hence, the device level model is unlikely to be absolutely identical to the model derived from typical parameters values found on OP AMP data sheets. From the simulation results the CMRR(0) values are approximately (1) macromodel 90 dB, (2) UA741 transistor model 101 dB. Similarly, the common-mode zero frequencies are approximately (1) macromodel 200 Hz, (2) UA741 transistor model 500 Hz.

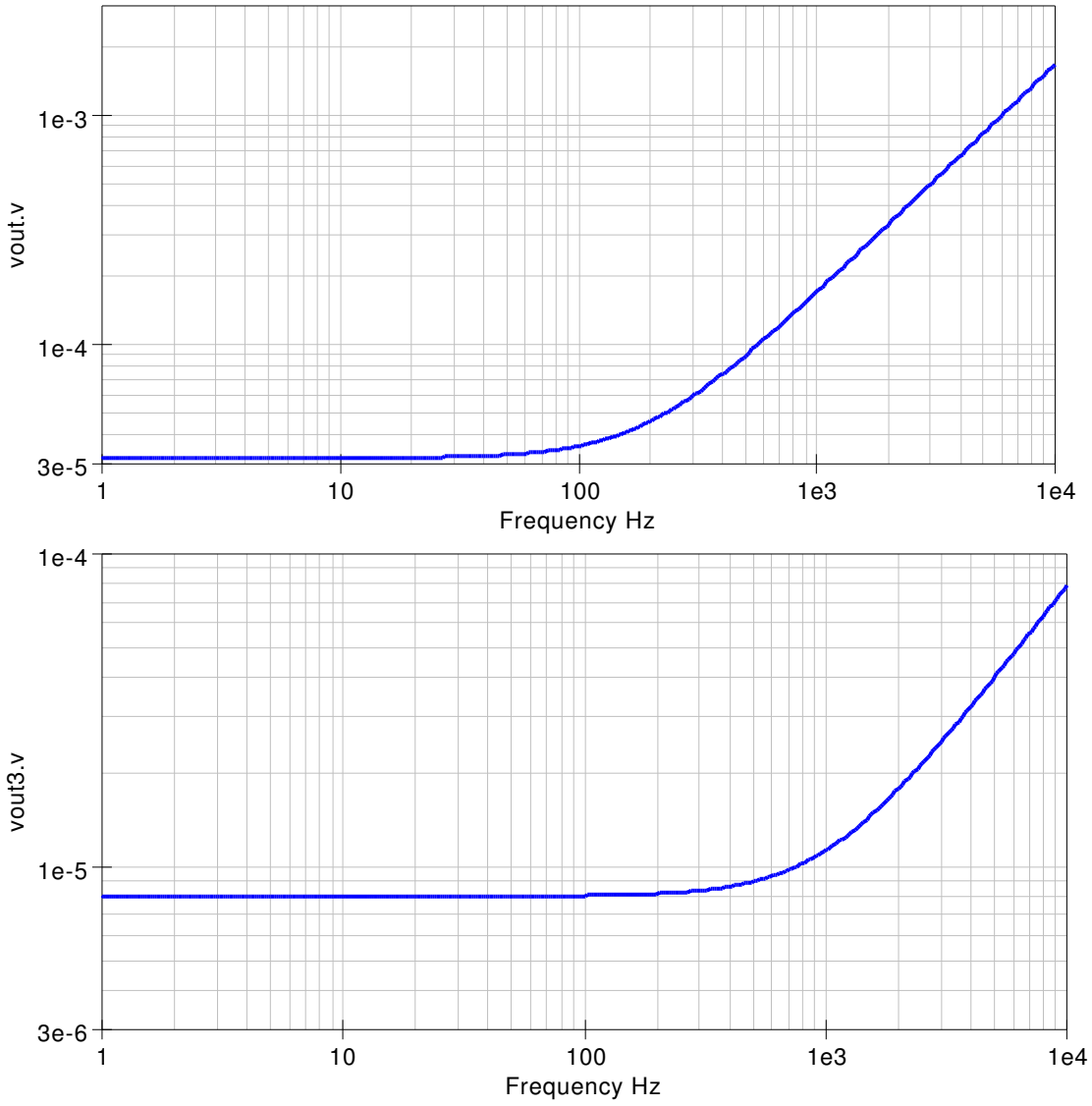


Figure 7.21: Simulation test results for the circuit shown in Fig. 7.20.

7.8 Large signal transient domain OP AMP macromodels

The modular macromodel introduced in the previous sections concentrated on modelling OP AMP performance in the small signal AC domain. Large signal models need to take into account the passage of signals through an OP AMP in the time domain and limit the excursion of voltage and current swings to the practical values found in actual amplifiers. Starting with the AC domain macromodel introduced in the previous sections, adding a slew rate limiting stage and a overdrive stage will more correctly model OP AMP high speed large signal limitations. Furthermore, by adding output voltage and current limiting stages the OP AMP macromodel will correctly model large signal effects when signal levels approach circuit power supply voltages or the OP AMP output current limits.

7.8.1 Slew rate macromodel derivation

The slew rate of an OP AMP can be modelled by limiting the current charging $CP1$ in the first voltage gain stage POLE1. From Fig. 7.9

$$GMP1\,(V(IN\_P1) - V(IN\_N1)) = \frac{V(POLE\_1\_OUT1)}{RADO} + CP1 * \frac{dV(POLE\_1\_OUT1)}{dt} \tag{7.14}$$

Hence, provided $RADO$ is large[7]

$$GMP1\,(V(IN\_P1) - V(IN\_N1)) \simeq CP1 * \frac{dV(POLE\_1\_OUT1)}{dt} \tag{7.15}$$

But $CP1 = \dfrac{1}{2\pi * GBP}$
Yielding

$$GMP1\,(V(IN\_P1) - V(IN\_N1)) \simeq \frac{1}{2\pi * GBP} * \frac{dV(POLE\_1\_OUT1)}{dt} \tag{7.16}$$

Moreover, if $\dfrac{dV(POLE\_1\_OUT1)}{dt}$ is set equal to the OP-AMP slew rate then the current charging *CP1* will be limited to the maximum allowed. In Fig. 7.9 $GMP1$ is 1 S.

Therefore, voltage difference $V(IN\_P1) - V(IN\_N1)$

must be set to $\dfrac{1}{2\pi * GBP} * \dfrac{dV(POLE\_1\_OUT1)}{dt}$.

This is done by the network SLEWRT shown in Fig. 7.22, where

[7]This condition is normally true because *RADO* is set to the DC open loop differential gain in macromodule POLE1.

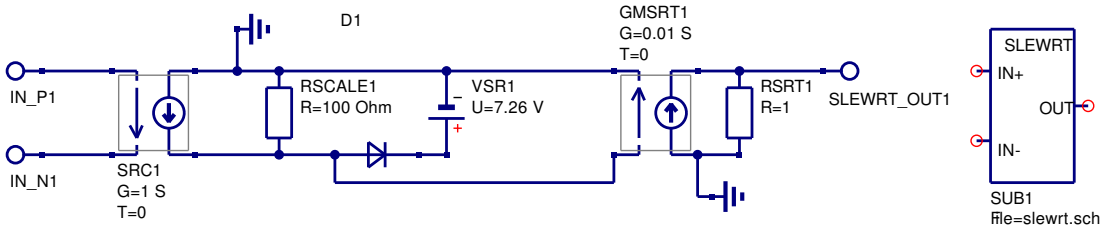


Figure 7.22: OP AMP slew rate macromodel.

1. *RSCALE1* = 100 Ω = Scaling resistance (Scale factor x 100).
2. *SRC1* G = 1 S.
3. *VSR1* = V1.
4. *GMSRT1* G = 0.01 S. (Scale factor = 1/100).
5. *RSRT1* = 1 Ω

And,

1. $V1 = \dfrac{100 * Positive\_slew\_rate}{2\pi * GBP} - 0.7V$
2. $V2 = \dfrac{100 * Negative\_slew\_rate}{2\pi * GBP} - 0.7V$
3. The diode parameters are IS=1e-12 IBV=20mA BV=V1+V2, others default.

Typical values for the UA741 OP AMP are:

1. $Positive\_slew\_rate = Negative\_slew\_rate = 0.5\text{V}/\mu\text{S}$.
2. $V1 = V2 = 7.25\text{V}$.

Scaling is used in the slew rate model to allow the use of higher voltages in the clamping circuit. Increased voltages reduce errors due to the forward biased junction voltage. Current limiting results by clamping the voltage across resistor $RSCALE1$ with a diode. This diode acts as a zener diode and saves one nonlinear junction when compared to conventional clamping circuits. The output section of the SLEWRT circuit removes the internal scaling yielding an overall gain of unity for the module.

The circuit in Fig. 7.23 demonstrates the effect of slew rate limiting on OP AMP transient performance. Three identical OP AMP inverter circuits are driven from a common input 10 kHz AC signal source. Voltage controlled voltage sources are used to amplify the input signal to the second and third circuits. The three input signals are (1) 5 V peak, (2) 10 V peak and (3) 15 V peak respectively. The input and output waveforms for this circuit are illustrated in Fig. 7.24. The effect of slew rate limiting on large signal transient performance is clearly demonstrated by these curves. In the case of the 15 V peak input signal the output signal (vout3.Vt) has a slope that is roughly 0.5 V per μS.

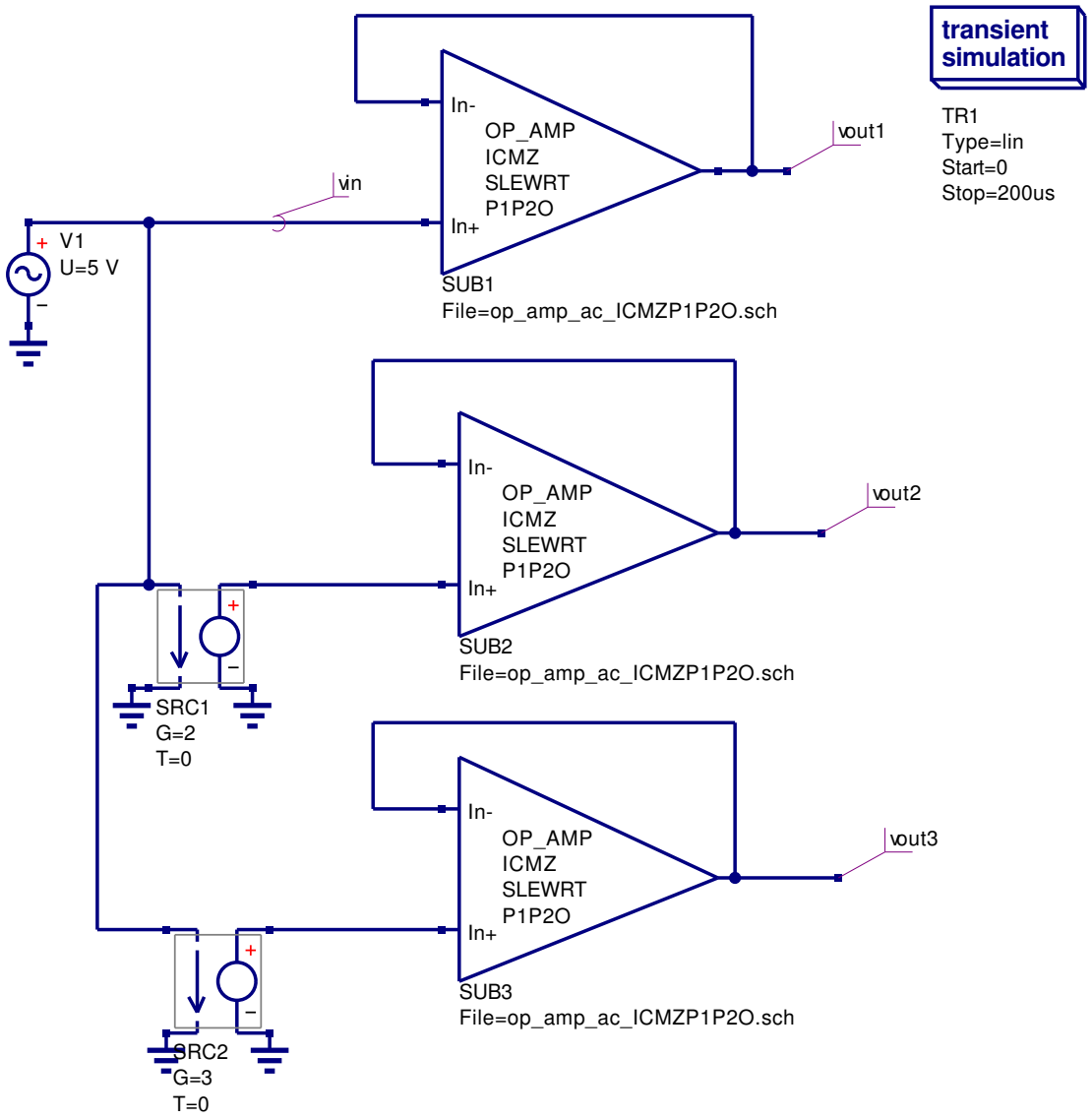


Figure 7.23: OP AMP slew rate test circuit.

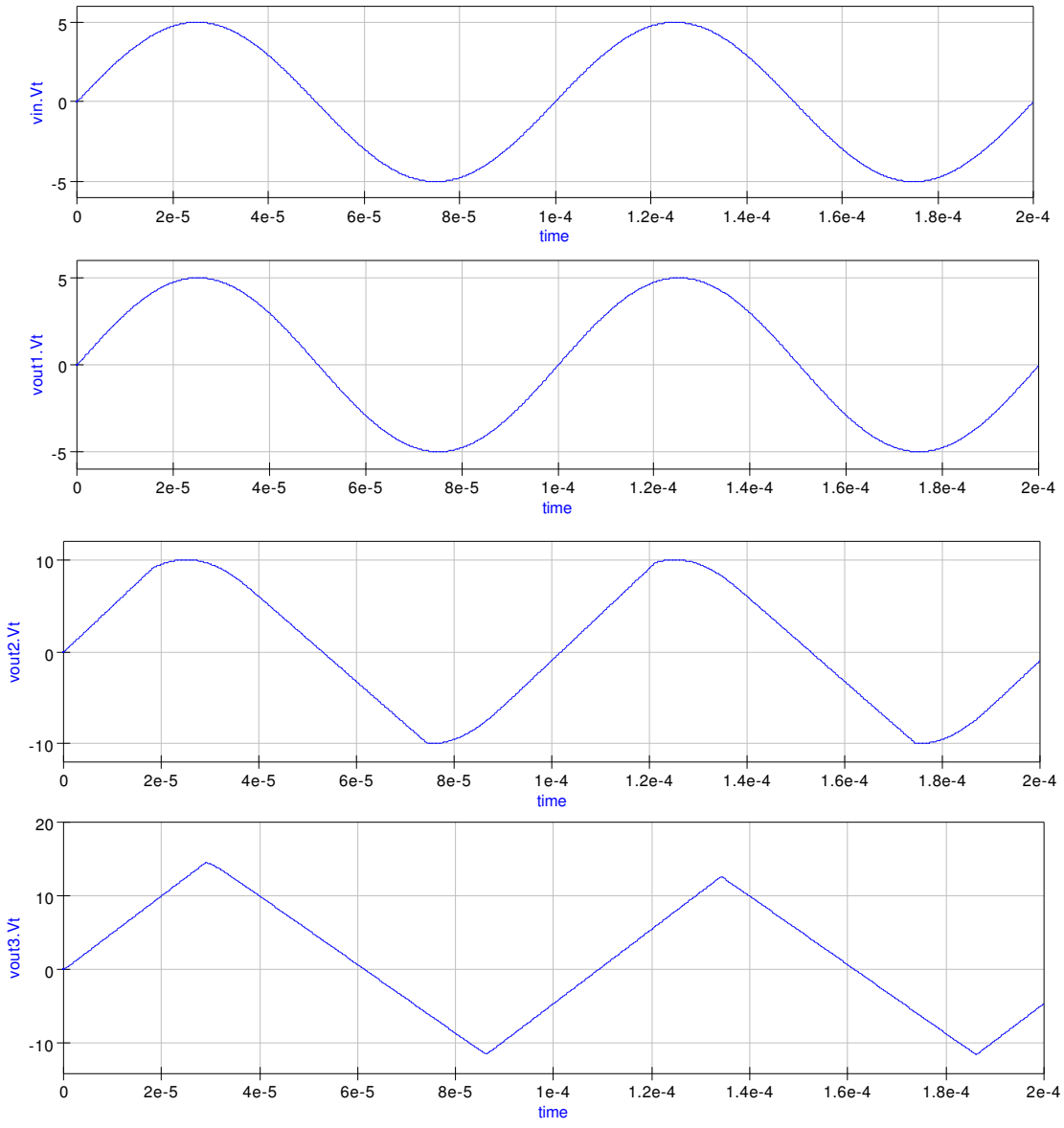


Figure 7.24: OP AMP slew rate simulation waveforms for the circuit shown in Fig. 7.23.

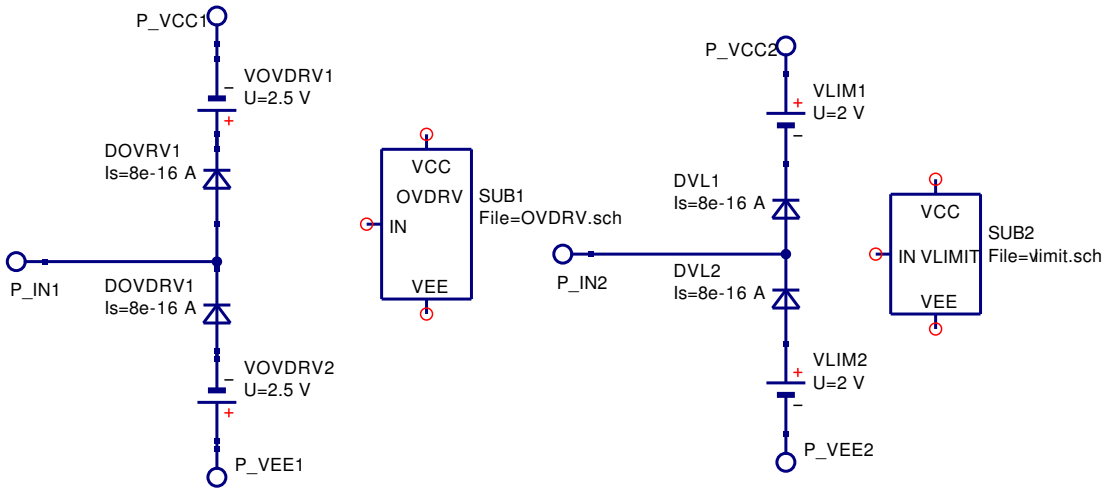


Figure 7.25: OP AMP overdrive and output voltage limiting macromodels.

7.8.2 Modelling OP AMP overdrive and output voltage limiting

Large transient signals can overdrive an OP AMP causing it's output voltage to saturate. On removal of the overdrive signal an OP AMP takes a finite time to recover[8] and return to normal linear circuit behaviour. When saturated the output voltage is clamped at a voltage close to the plus or minus power rail voltage. The overdrive and voltage clamping properties of an OP AMP are related and macromodels for both effects need to be added to an OP AMP model when simulating OP AMP overdrive characteristics. However, in many circuit simulations the overdrive macromodel can be left out without loss of functionality or accuracy.

The effect of overdrive signals can be modelled by a voltage clamping circuit which takes account of OP AMP recovery time from voltage overdrive. This extra element clamps the output of the POLE1 module at a level above the OP AMP DC supply voltages. The overall effect of the overdrive circuit is to delay the restoration of linear circuit behaviour when an overload signal is removed. In contrast to the overdrive module the output voltage limiting module clamps the output voltage to a voltage close to the power rail voltages, clipping any output voltage excursions above the power rail voltage levels. Figure 7.25 illustrates the macromodels for the overdrive and output voltage limiting models, where

1. *VOVDR1* = 2.5 V = (Positive slew rate)*(Amplifier recovery time).
2. *VOVDR2* = 2.5 V = (Negative slew rate)*(Amplifier recovery time).
3. *VLIM1* = 2.0 V = (+ supply voltage) - (Maximum positive output voltage) + 1 V.

[8]Overload recovery time of an OP AMP is the time required for the output voltage to recover to a rated output voltage from a saturated condition. Typical values are in the μ S region.

4. *VLIM2* = 2.0 V = (- supply voltage) - (Maximum negative output voltage) + 1 V.

5. The diode parameters are Is = 8e-16 A, others default.

Typical values for the UA741 OP AMP are:

1. Amplifier recovery time 5 μS.

2. + supply voltage = 15 V.

3. - supply voltage = -15 V.

4. Maximum positive output voltage = 14 V.

5. Maximum negative output voltage = -14 V.

The test circuit given in Fig. 7.26 illustrates the effects of signal overdrive and output voltage clamping on a unity gain buffer circuit. The test input signal is a 1 kHz signal with the following drive voltages (1) vin1 = 10 V peak, (2) vin2 = 18 V peak, and (3) vin3 = 22 V peak. The corresponding output waveforms are shown in Fig. 7.27. These indicate that increasing overdrive signals results in longer OP AMP recovery times before the amplifier returns to linear behaviour.

7.8.3 Modelling OP AMP output current limiting

Most general purpose OP AMPs have a network at the circuit output to protect the device from high load currents generated by shorting the output terminal to ground or some other situation where a high current flows through the OP AMP output stage. The electrical network shown in Fig. 7.28 acts as a current limiter: current flowing between pins `P_IN1` and `P_OUT1` is sensed by current controlled voltage generator HCL1. The voltage output from generator HCL1 is in series with voltage controlled generator ECL1. The connection of these generators is in opposite polarity. Hence, when the load current reaches the maximum allowed by the OP AMP either diode DCL1 or DCL2 turns on clamping the OP AMP output voltage preventing the output current from increasing. The parameters for the current limiter macromodel are given by

1. *RDCL1* = 100 MΩ = Dummy resistor.

2. *ECL1* G = 1.

3. *HCL1* G = 36Ω = 0.9 V/(Maximum output current A).

4. The diode parameters are Is = 1e-15 A, others default.

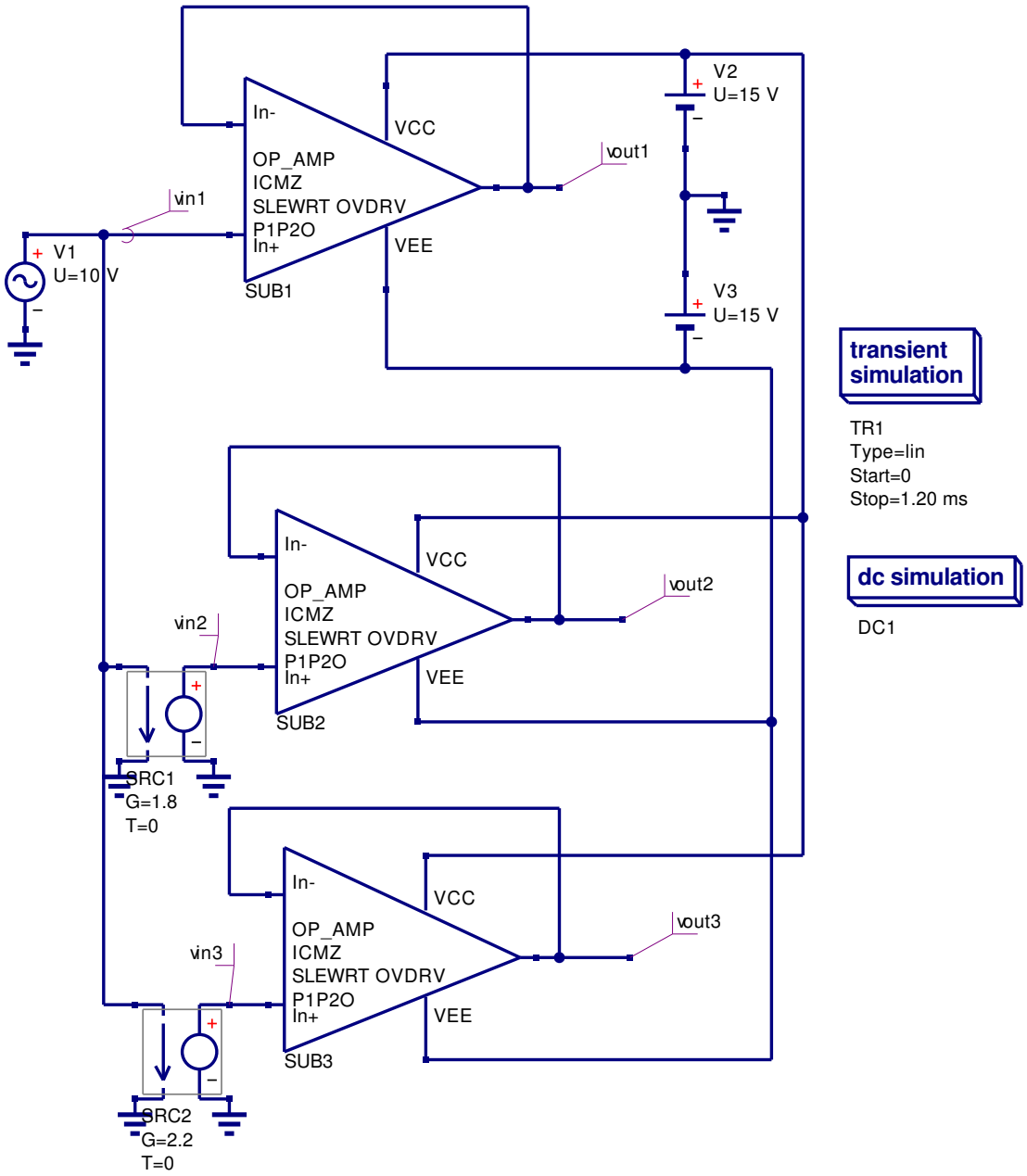


Figure 7.26: OP AMP overdrive and output voltage limiting test circuit.

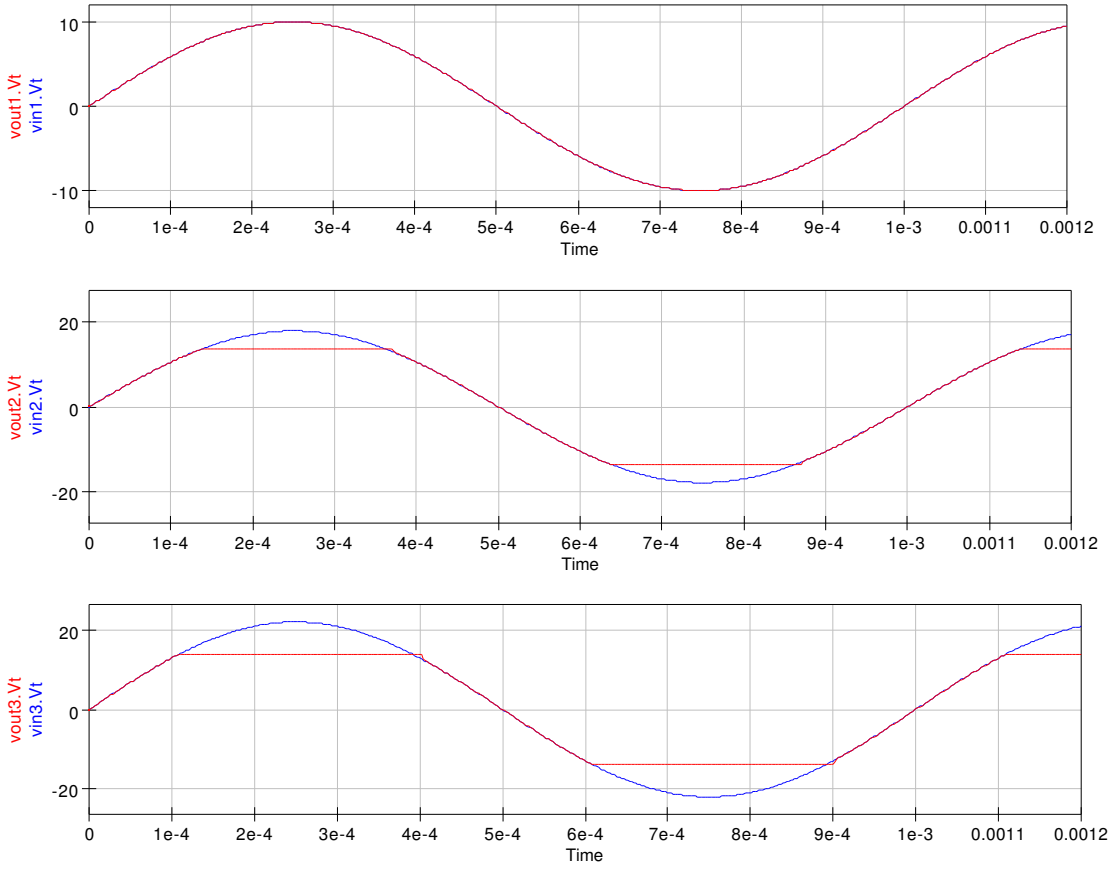


Figure 7.27: OP AMP overdrive and output voltage limiting waveforms for the circuit shown in Fig. 7.26.

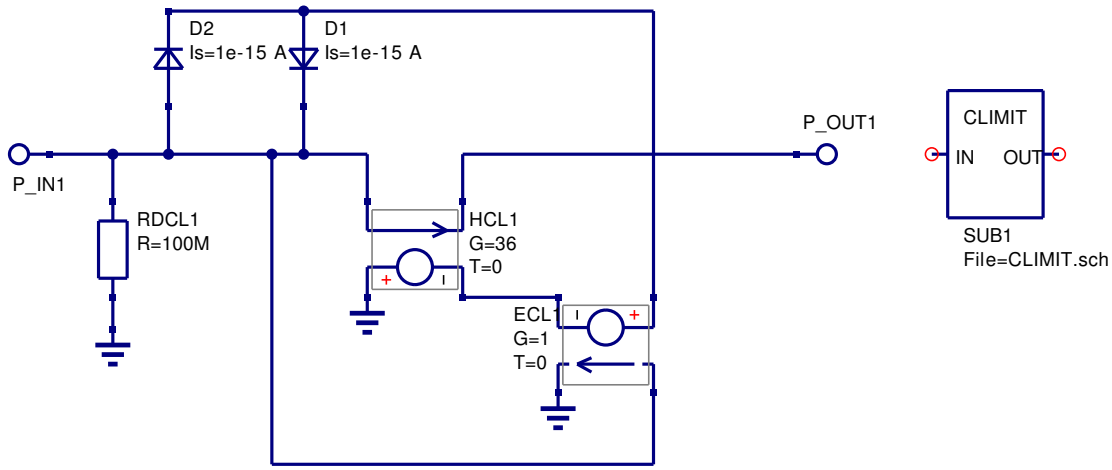


Figure 7.28: OP AMP output current limiter macromodel.

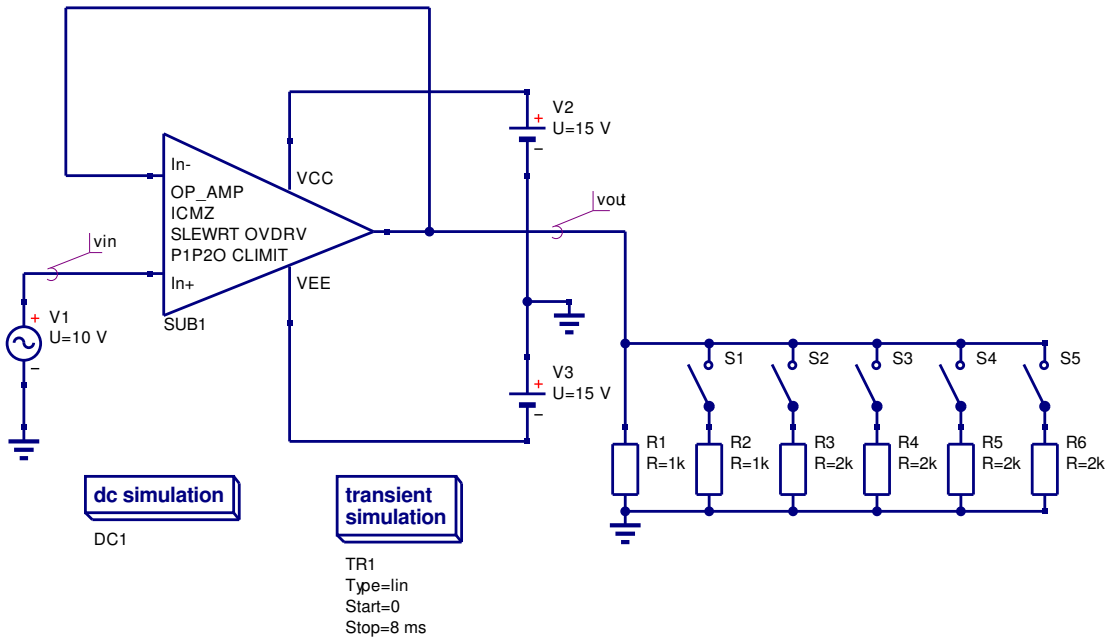


Figure 7.29: OP AMP output current limiter test circuit.

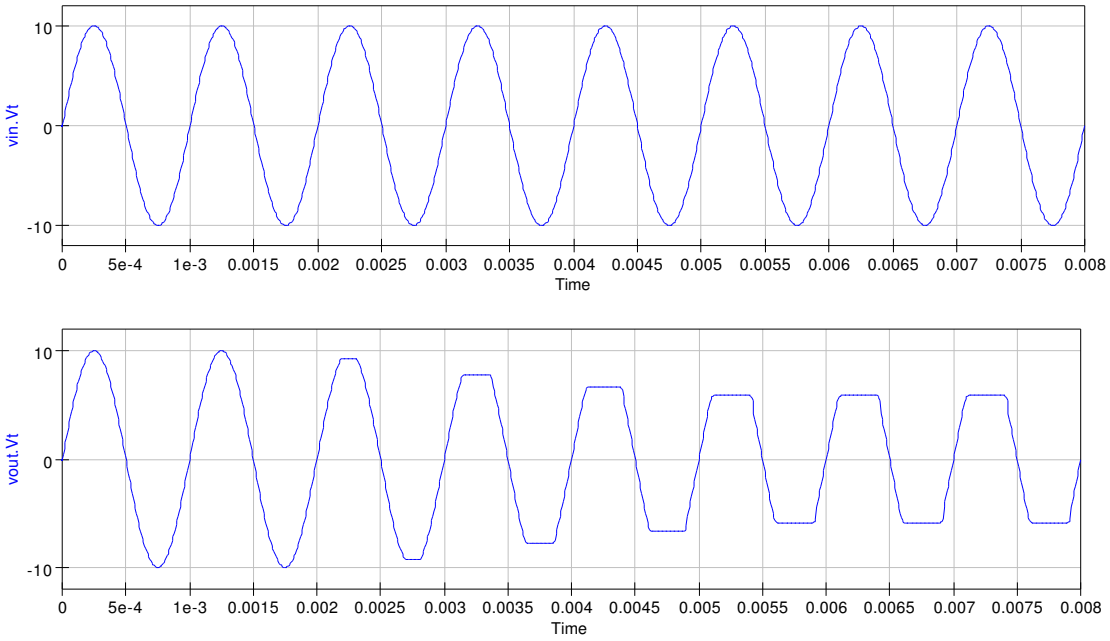


Figure 7.30: Simulation waveforms for current limiter test circuit shown in Fig. 7.29.

Parameter	UA741	OP27	OP42	OPA134	AD746	AD826
Offset voltage (V)	7e-4	30e-6	4e-4	5e-4	3e-4	5e-4
Bias current (A)	80e-9	15e-9	130e-12	5e-12	110e-12	3-3e-6
Offset current (A)	20e-9	12e-9	6e-12	2e-12	45e-12	25e-9
Differential input res. (ohm)	2e6	4e6	1e12	1e13	2e11	300e3
Differential input cap. (F)	1.4e-12		6e-12	2e-12	5.5e-12	1.5e-12
Avd(0) dB	106	125	120	120	109	75
fp1 (Hz)	5	6	20	5	0.25	10e3
fp2 (Hz)	3e6	17e6	20e6	10e6	35e6	100e6
CMRR(0) dB	90	125	96	100	85	100
fcm (Hz)	200	2e3	100e3	500	3e3	2e3
GBP (Hz)	1e6	8e6	10e6	8e6	13e6	35e6
Rout (ohm)	75	70	50	10	10	8
Slew rate (V per micro sec.)	0.5	2.8	50	20	75	300
Overdrive recovery time (S)	5e-6		700e-9	0.5e-6		
DC supply current (A)	1.4e-3	2.5e-3	5.1e-3	4e-3	7e-3	6.6e-3
Short circuit output current(A)	34e-3	32e-3	30e-3	40e-3	25e-3	90e-3
Common-mode input res. (ohm)	1.3e8	2e9		1e13	2.5e11	
Common-mode input cap. (F)				5e-12	5.5e-12	

Table 7.1: Typical OP AMP parameters taken from device data sheets.

A typical value for the UA741 OP AMP short circuit current is 34 mA at 25^{o}C.

Figures 7.29 and 7.30 show a simple current limiter test circuit and the resulting test waveforms. In this test circuit time controlled switches decrease the load resistors at 1 mS intervals. When the load current reaches roughly 34 mA the output voltage is clamped preventing further increases in load current.

7.9 Obtaining OP AMP macromodel parameters from published device data

The OP AMP modular macromodel has one very distinct advantage when compared to other amplifier models namely that it is possible to derive the macromodel parameters directly from a common set characteristics found on the majority of manufacturer's data sheets. The data given in Table. 8.1 shows a typical range of values found on OP AMP data sheets. In cases where a particular parameter is not given then a starting point is to use a value obtained from a data sheet of an equivalent device. The macromodel element values are then calculated using the equations presented in the previous sections of this tutorial. As a rule of thumb it is good practice to test each block in the modular macromodel prior to constructing a complete OP AMP macromodel.

7.10 More complete design examples.

In this section two larger design examples are presented. These demonstrate the characteristics of the various OP AMP macromodels introduced in the previous text and attempt to give readers guidance as to the correct model to choose for a particular simulation.

7.10.1 Example 1: State variable filter design and simulation

The circuit given in Fig. 7.31 is a state variable filter which simultaneously generates band-pass, high-pass and low-pass responses. The circuit consists of an OP AMP adder and two integrator circuits and requires three OP AMPS, two capacitors and a number of resistors. The selection of the type of OP AMP for successful operation of this filter is critical because devices with high offset voltage will cause the integrators to saturate and the circuit will not function correctly. For operation below 20 kHz the OP27 is a good choice of OP AMP because of it's low offset voltage in the μV region. In this simulation both the DC characteristics and small signal AC transfer characteristics are needed to check the filter design, hence the AC macromodel with the DC parameters embedded in the input stage should allow accurate modelling of the filter performance.[9] The insert in Fig. 7.31 list the DC output voltages for each of the OP AMP stages indicating that the integrators are not saturated. The design of the state variable filter uses the following equations:

1. The superposition principle yields

$$vhp = -\frac{R1}{R6}vin - \frac{R1}{R7}vlp + \left(1 + \frac{R1}{R7 \parallel R6}\right)\frac{R4}{R4 + R5}vbp \tag{7.17}$$

When $R1 = R6 = R7$

[9]The magnitude of the output signals from the filter should also be checked to ensure that these signals do not exceed the power supply voltages.

$$vhp = -vin - vlp + \frac{3R4}{R4 + R5} vbp \tag{7.18}$$

2. Also

$$vbp = -\frac{1}{j\frac{f}{f_0}} vhp \tag{7.19}$$

where

$$f_0 = \frac{1}{2\pi R_2 C_1} = \frac{1}{2\pi R_3 C_2} \tag{7.20}$$

3. Similarly

$$vlp = -\frac{1}{j\frac{f}{f_0}} vbp = -\frac{1}{(\frac{f}{f_0})^2} vhp \tag{7.21}$$

4. Hence

$$\frac{vhp}{vin} = \frac{(\frac{f}{f_0})^2}{1 - (\frac{f}{f_0})^2 + (\frac{j}{Q})(\frac{f}{f_0})} \tag{7.22}$$

Where

$$Q = \frac{1}{3}(1 + \frac{R5}{R4}) \tag{7.23}$$

5. Also

$$\frac{vbp}{vin} = \frac{j\frac{f}{f_0}}{1 - (\frac{f}{f_0})^2 + (\frac{j}{Q})(\frac{f}{f_0})} \tag{7.24}$$

6. Also

$$\frac{vlp}{vin} = \frac{-1}{1 - (\frac{f}{f_0})^2 + (\frac{j}{Q})(\frac{f}{f_0})} \tag{7.25}$$

Assuming $f_0 = 1$ kHz and the required bandwidth of the band pass filter is 10 Hz, on setting $R1 = R6 = R7 = 47k\Omega$ and $C1 = C2 = 2.2nF$, calculation yields $R2 = R3 = 72.33k\Omega$[10] In this design $Q = 1k/10 = 100$. Hence setting $R4 = 1k\Omega$ yields $R5 = 294k\Omega$ (1 % tolerance). The simulation waveforms for the band pass output are given in Fig. 7.32 [11]. When the circuit Q factor is reduced to lower values the other filter outputs act as traditional high and low pass filters. The simulation results for Q factor one are shown in Fig. 7.33.

[10]The values of $R2$ and $R3$ need to be trimmed if the filter center frequency and bandwidth are required to high accuracy.

[11]Note that the input signal *vin* has been set at 0.1 V peak. The circuit has a Q factor of 100 which means that the band pass output voltage is 10 V peak. Input signals of amplitude much greater than 0.1 V are likely to drive the output signal into saturation when the power supply voltages are $\pm 15V$.

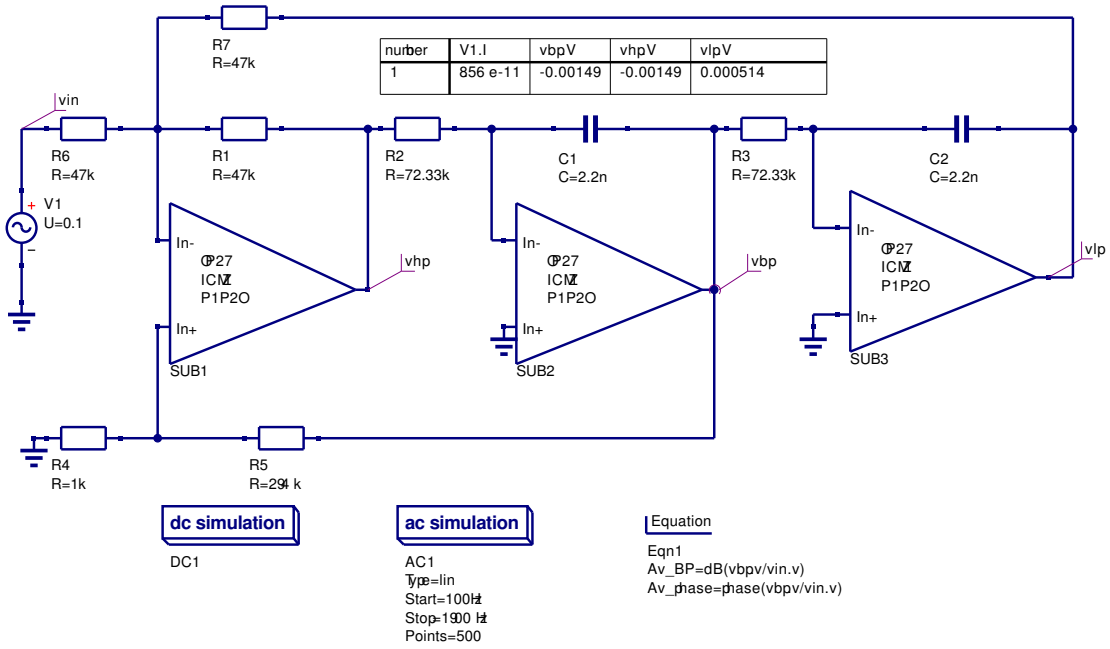


Figure 7.31: Three OP AMP state variable filter.

7.10.2 Example 2: Sinusoidal signal generation with the Wien bridge oscillator

The Wien bridge sinusoidal oscillator has become a classic due to it's simplicity and low distortion capabilities. It is an ideal vehicle for demonstrating the properties of OP AMP macromodels and indeed the performance of circuit simulators. Shown in Fig. 7.34 is the basic Wien bridge oscillator which consists of a single OP AMP with negative and positive feedback circuits. The design equations for this circuit are

1. Non-inverting amplifier.
$$\frac{vout}{v+} = 1 + \frac{R3}{R4} \tag{7.26}$$

2. Feedback factor
$$b = \frac{vout}{v+} = \frac{1}{3 + j(\frac{f}{f_0} - \frac{f_0}{f})} \tag{7.27}$$
Where $f_0 = \dfrac{1}{2\pi R1C1} = \dfrac{1}{2\pi R2C2}$

3. Loop gain

The oscillator loop gain bA_v must equal one for stable oscillations. Hence,
$$bA_v = \frac{1 + \frac{R3}{R4}}{3 + j(\frac{f}{f_0} - \frac{f_0}{f})} \tag{7.28}$$

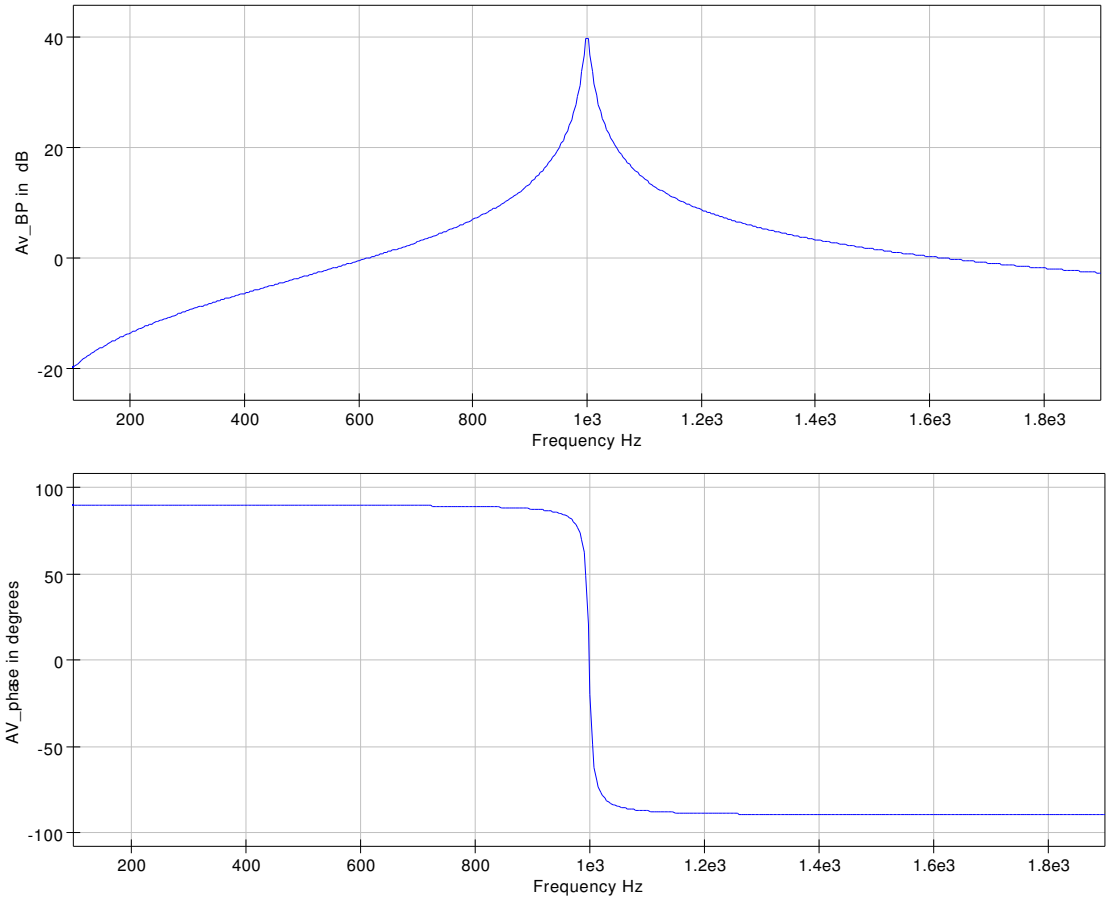


Figure 7.32: Simulation waveforms for current state variable filter circuit shown in Fig. 7.31.

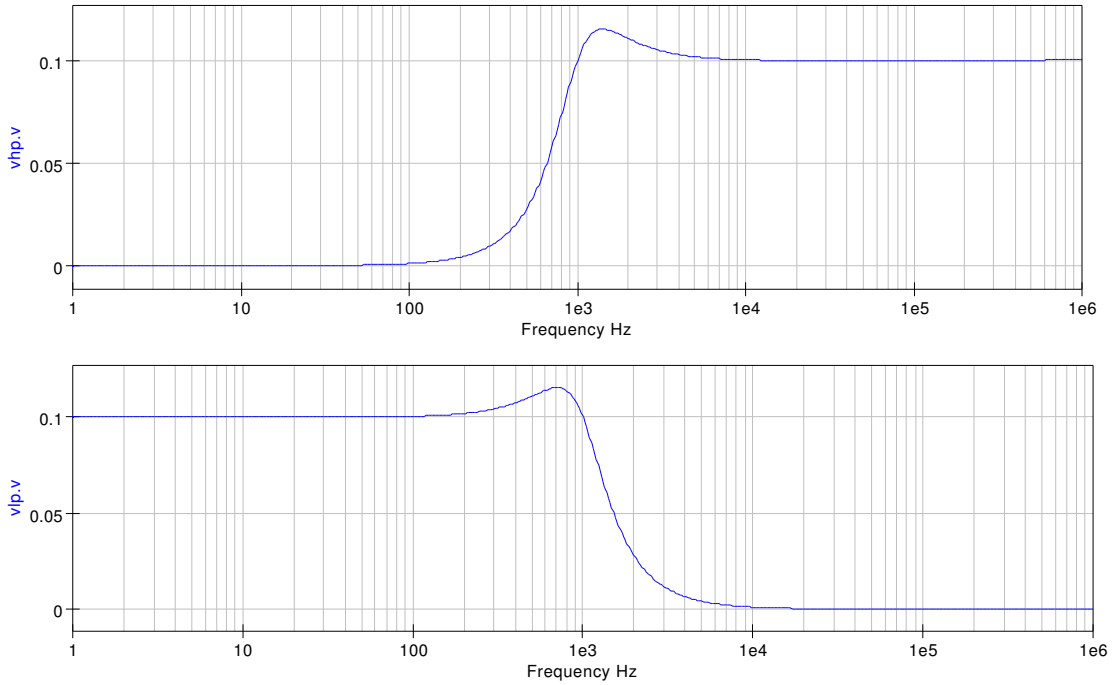


Figure 7.33: State variable low pass and high pass response for $Q = 1$, $R5 = 2k\Omega$.

Moreover, at $f = f_0$,

$$bA_v = \frac{1 + \frac{R3}{R4}}{3} \tag{7.29}$$

Setting $R3/R4$ slightly greater than two causes oscillations to start and increase in amplitude during each oscillatory cycle. Furthermore, if $R3/R4$ is less than two oscillations will never start or decrease to zero.

Figure 7.35 shows a set of Wien bridge oscillator waveforms. In this example the OP AMP is modelled using the OP27 AC macromodel. This has been done deliberately to demonstrate what happens with a poor choice of OP AMP model. The oscillator frequency is 10 kHz with both feedback capacitors and resistors having equal values. Notice that the oscillatory output voltage continues to grow with increasing time until it's value far exceeds the limit set by a practical OP AMP power supply voltages. The lower of the two curves in Fig. 7.35 illustrates the frequency spectrum of the oscillator output signal. The data for this curve has been generated using the Time2Freq function. Adding slew rate and voltage limiting to the OP27 macromodel will limit the oscillator output voltage excursions to the OP AMP power supply values. The waveforms for this simulation are shown in Fig. 7.36. When analysing transient response data using function Time2Freq it is advisable to restrict the analysis to regions of the response where the output waveform has reached a steady state otherwise the frequency spectrum will include effects due to growing, or decreasing, transients. The voltage limiting network clips the oscillator output voltage restricting its excursions to below the OP AMP power supply voltages. The clipping is

very visible in Fig. 7.36. Notice also that the output waveform is distorted and is no longer a pure sinusoidal waveform of 10 kHz frequency. Odd harmonics are clearly visible and the fundamental frequency has also decreased due to the signal saturation distortion. In a practical Wien bridge oscillator the output waveform should be a pure sinusoid with zero or little harmonic distortion. One way to achieve this is to change the amplitude of the OP AMP gain with changing signal level: as the output signal increases so Av is decreased or as the output signal level decreases Av is increased. At all times the circuit parameters are changed to achieve the condition $bAv = 1$. The circuit shown in Fig. 7.37 uses two diodes and a resistor to automatically change the OP AMP closed loop gain with changing signal level. Fig. 7.38 shows the corresponding waveforms for the Wien bridge circuit with automatic gain control. Changing the value of resistor $R5$ causes the amplitude of the oscillator output voltage to stabilise at a different value; decreasing $R5$ also decreases $vout$. The automatic gain control version of the Wien bridge oscillator also reduces the amount of harmonic distortion generated by the oscillator. This can be clearly observed in Fig. 7.38. Changing the oscillator frequency can be accomplished by either changing the capacitor or resistor values in the feedback network b. To demonstrate how this can be done using Qucs, consider the circuit shown in Fig. 7.39. In this circuit time controlled switches change the value of both capacitors as the simulation progresses. The recorded output waveform for this circuit is shown in Fig. 7.40.

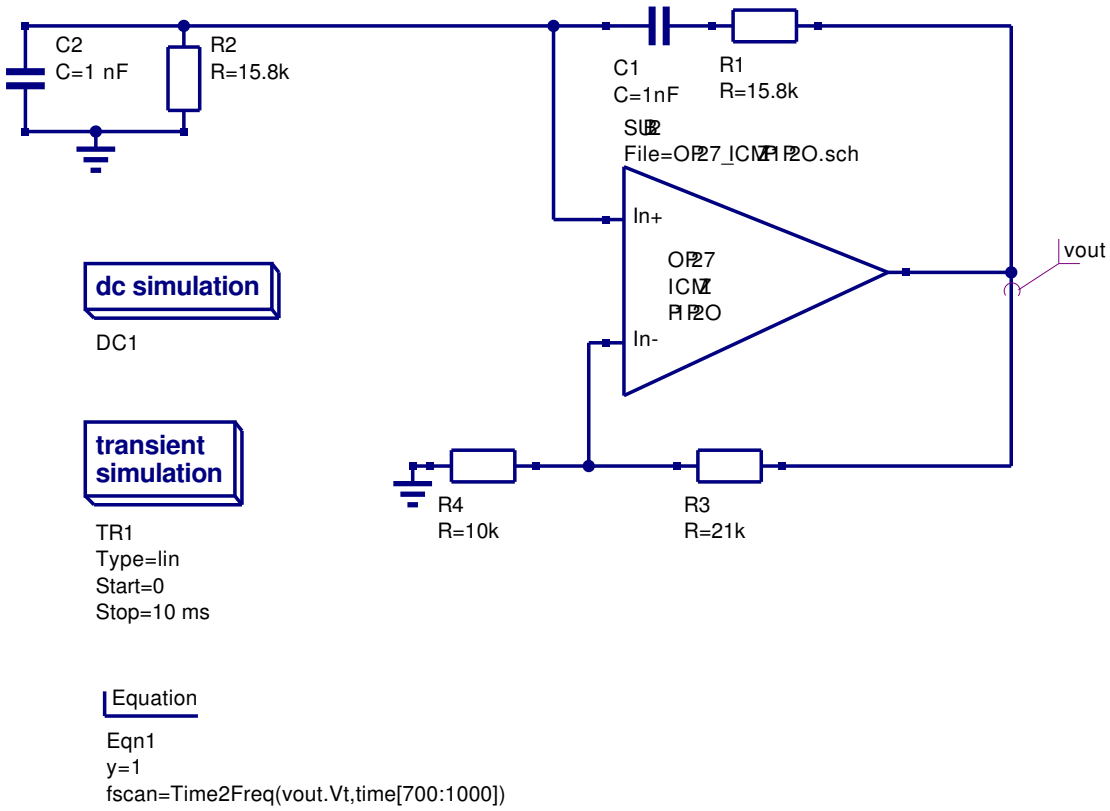


Figure 7.34: Classic Wien bridge sinusoidal oscillator.

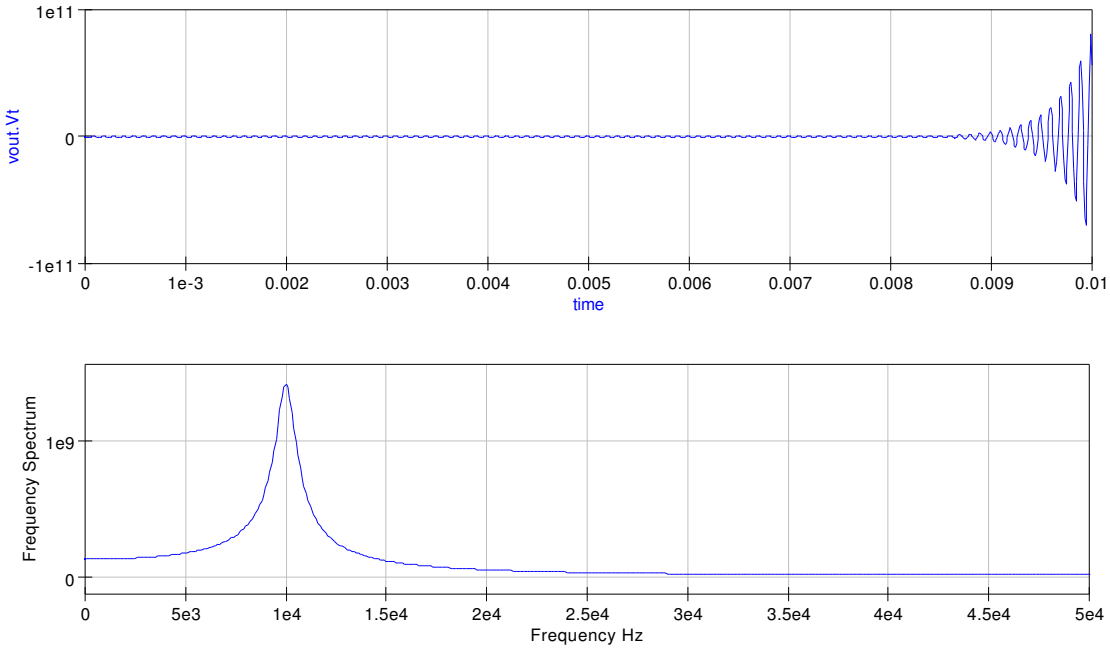


Figure 7.35: Simulation waveforms for the circuit shown in Fig. 7.34: OP27 AC macromodel.

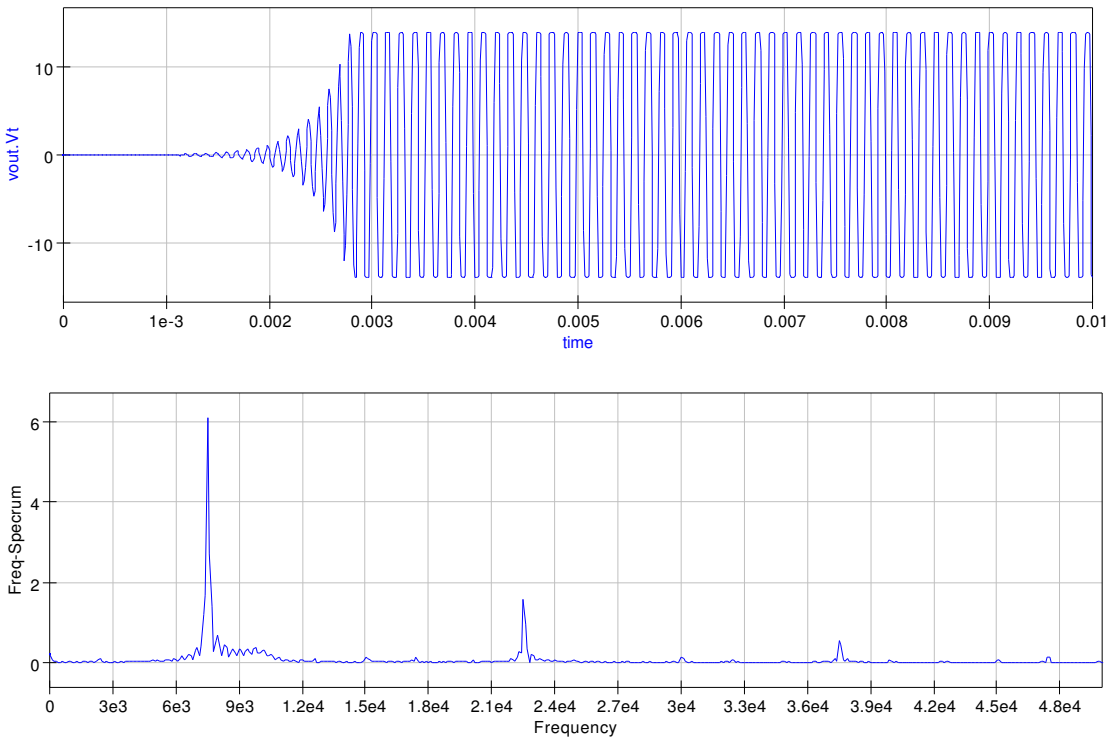


Figure 7.36: Simulation waveforms for the circuit shown in Fig. 7.34: OP27 AC + slew rate + vlimit macromodel.

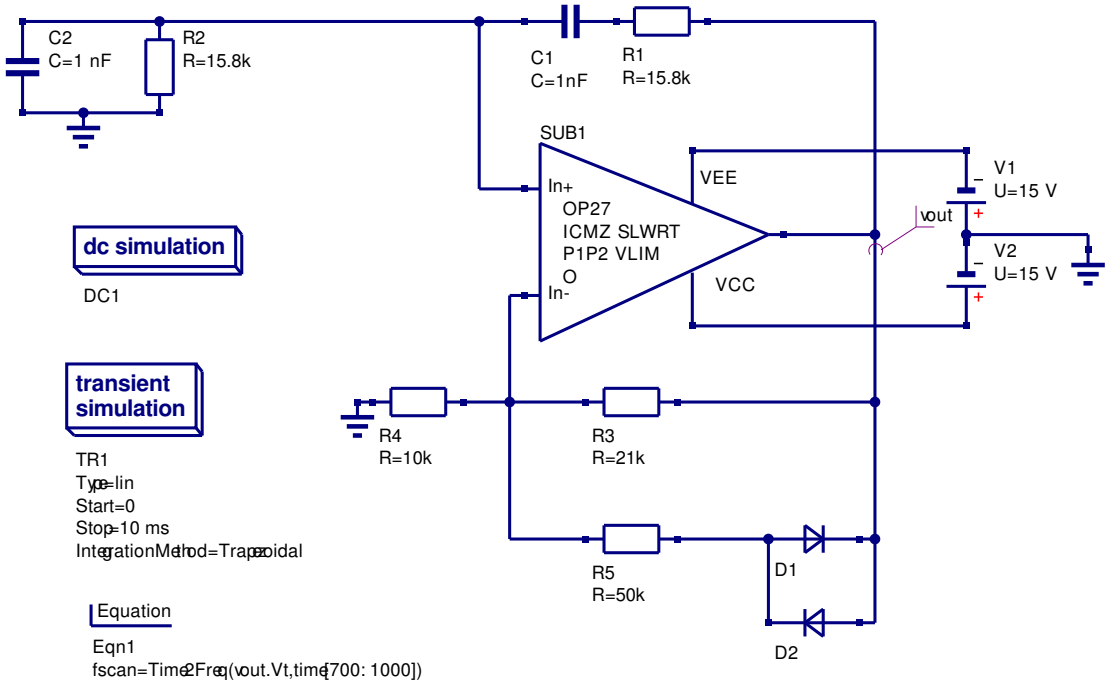


Figure 7.37: Wien bridge oscillator with automatic gain control.

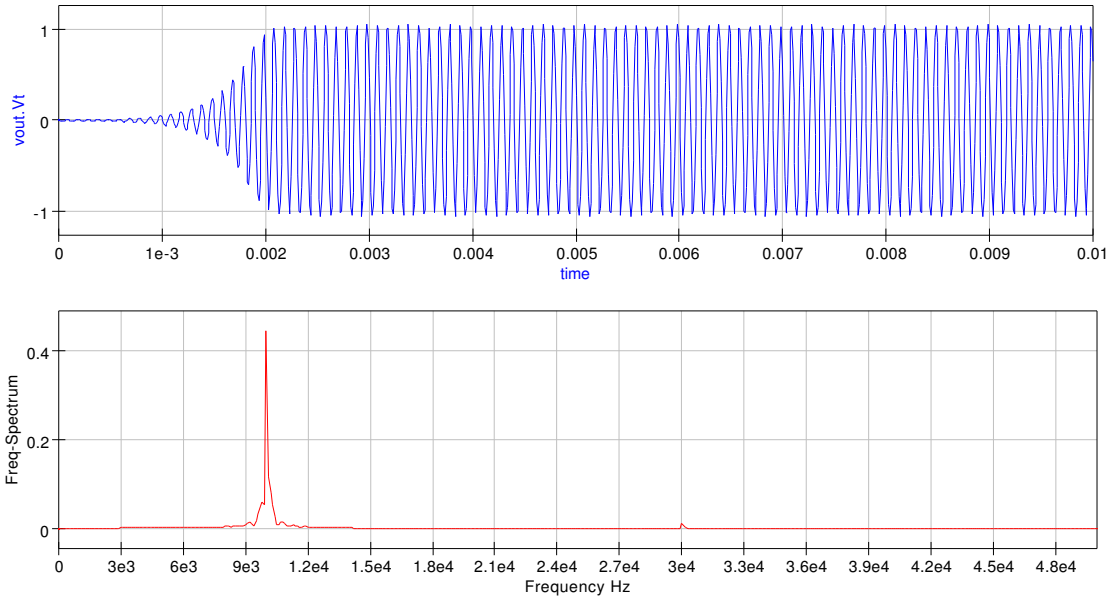


Figure 7.38: Simulation waveforms for the circuit shown in Fig. 7.37: OP27 AC + slew rate + vlimit macromodel.

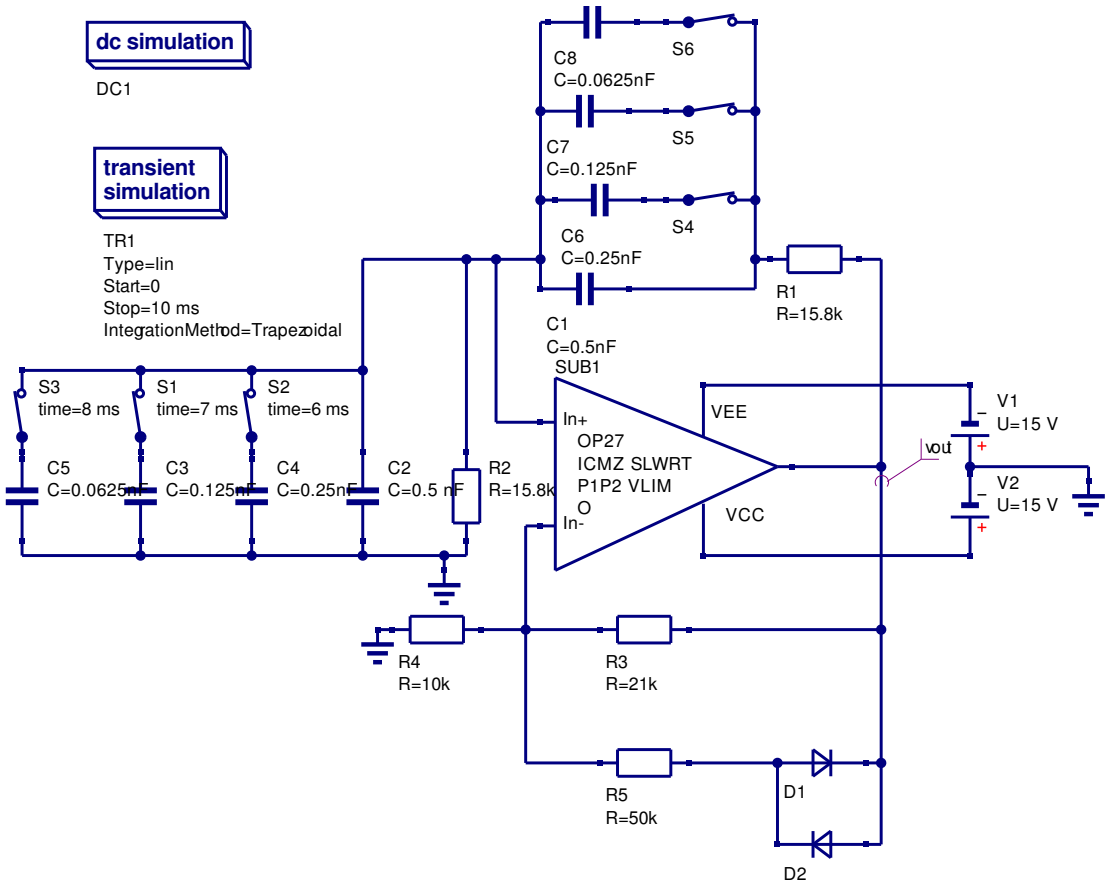


Figure 7.39: Wien bridge oscillator with switched capacitor frequency control.

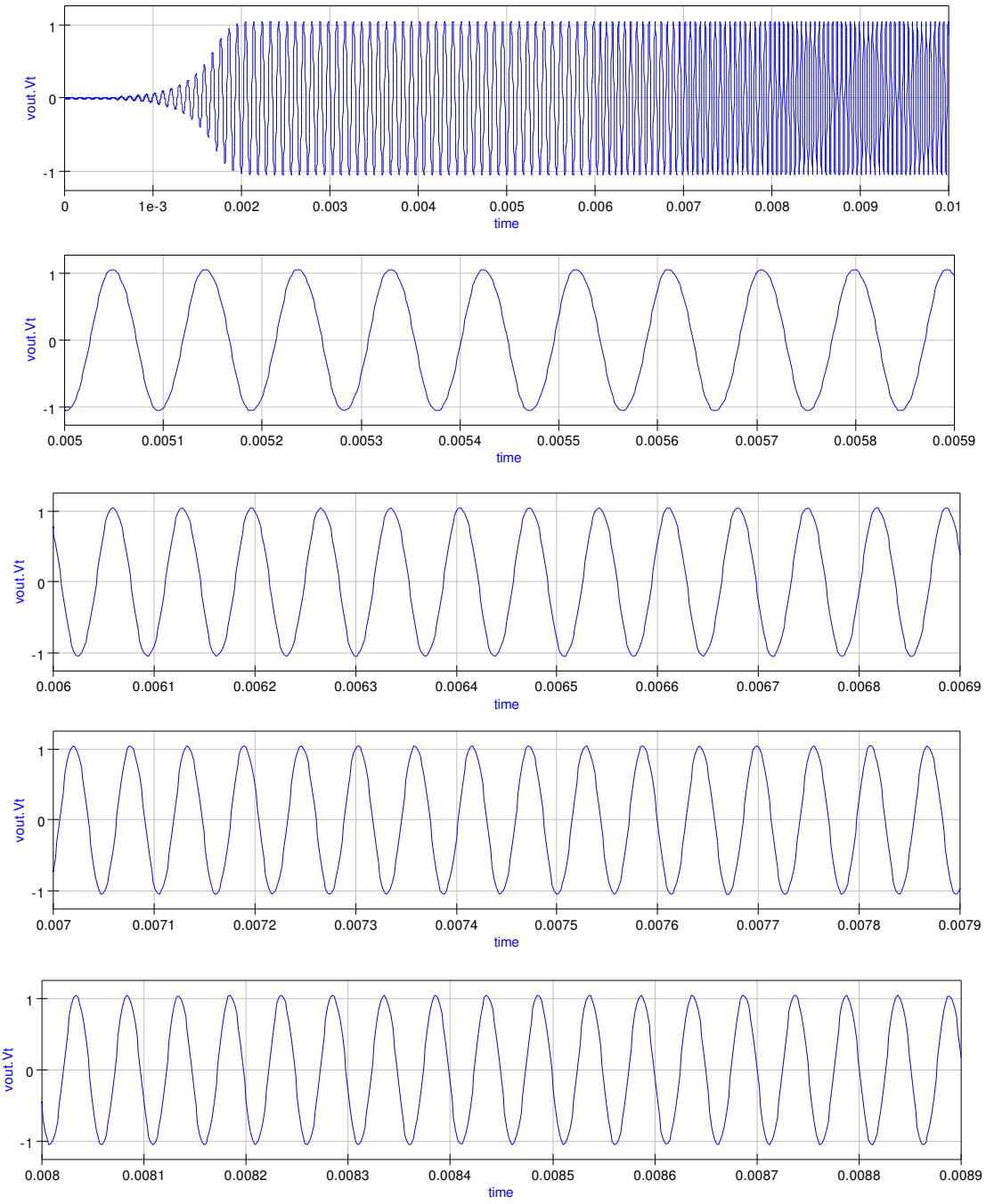


Figure 7.40: Simulation waveforms for the circuit shown in Fig. 7.39: OP27 AC + slew rate + vlimit macromodel.

7.11 Update number one: March 2007

In this first update to the operational amplifier tutorial readers will be introduced to Qucs macromodel model building using schematics and SPICE to Qucs conversion techniques, secondly to procedures for constructing Qucs operational amplifier libraries, and finally to two different approaches which allow existing OP AMP models to be extended to include new amplifier performance parameters, for example power supply rejection. This update is very much a report on the OP AMP modelling work that has been done by the Qucs development team since version 0.0.10 of the package was released in September 2006. Future Qucs releases will offer many significant improvements in OP AMP modelling particularly via SPICE to Qucs netlist conversion, subcircuit passing and equation embedding in Qucs schematics and library development. Following the release of Qucs 0.0.11, and a suitable period of time for new feature debugging, many of the ideas introduced in this update will be developed to include OP AMP model building using embedded equations in Qucs schematics.

7.11.1 Building a library component for the modular OP AMP macromodel

One of the main strengths of the modular macromodel approach to device modelling is the fact that the parameters implicit in each section of a macromodel are essentially independent, allowing subcircuit blocks to be easily connected together to form an overall device model. Taking this idea further one can construct a complete schematic for an OP AMP model from the circuitry that represents individual macromodel subcircuit blocks. The diagram shown in Fig. 7.41 illustrates a typical circuit schematic for a modular OP AMP macromodel. In this schematic the component values are for the UA741 OP AMP. By attaching a symbol to the modular macromodel schematic the UA741 modular OP AMP model is ready for general use and can be placed in an existing[12] or a user defined library. Moreover, by recalculating the component values further library elements can be constructed and the development of a more extensive Qucs OP AMP library undertaken[13].

7.11.2 Changing model parameters: use of the SPICEPP preprocessor

Changing the component data in Fig. 7.41 allows users to generate modular macromodels for different operational amplifiers. Although this is a perfectly viable approach to model generation it is both tedious and error prone. A more straightforward way is to get the

[12] Qucs 0.0.10, and earlier releases, were distributed with an OP AMP library called OpAmps. However, this only contained a component level model for the 741 OP AMP. Many of the models discussed in this text have been added to the Qucs OpAmps library. These should assist readers who wish to experiment with their own OP AMP circuits.

[13] One of the important future tasks is the development of component libraries for use with Qucs - this will take time but should be possible given enough effort by everyone interested in Qucs.

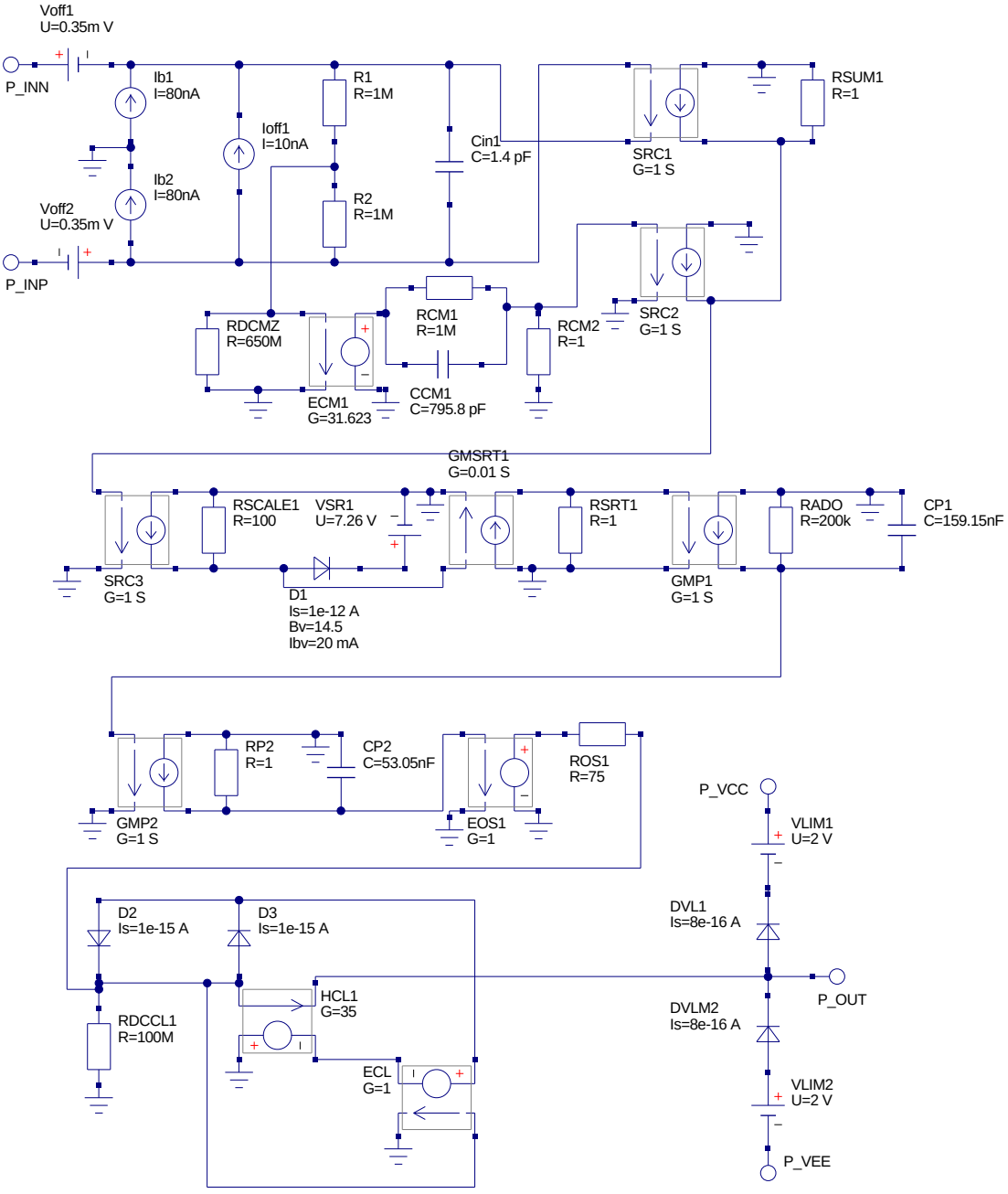


Figure 7.41: Modular OP AMP macromodel in schematic form - this model does not include signal overloading.

computer to do the tedious work involving component value calculation from device data. With this approach users are only required to enter the device data; as a simple list derived from manufacturers data sheets. One way to do this is to write a SPICE preprocessor template[14] and let a SPICE preprocessor generate the model for a specific OP AMP. The PS2SP template file for an OP27 OP AMP modular macromodel is given in Fig. 7.42. The resulting SPICE file is shown in Fig. 7.43. After construction of the SPICE OP27 netlist the Qucs OP27 model is generated via the schematic capture SPICE netlist facility.[15]

7.11.3 The Boyle operational amplifier SPICE model

The Boyle[16] operational amplifier model was one of the earliest attempts at constructing an OP AMP macromodel that achieved significantly reduced simulation times, when compared to those times obtained with discrete transistor level models[17], while maintaining acceptable functional properties and simulation accuracy. The Boyle macromodel was designed to model differential gain versus frequency, DC common-mode gain, device input and output characteristics, slew rate limiting, output voltage swing and short-circuit limiting. The circuit schematic for the Boyle macromodel of a bipolar OP AMP is illustrated in Fig. 7.44. This model consists of three connected stages: the input stage, the intermediate voltage gain stage and the output stage. Calculation of individual component values is complex, relying on a set of equations derived from the physical properties of the semiconductor devices and the structure of the electrical network. These equations are derived in the Boyle paper and summerised in the following list. Starting with I_{S1} =8.0e-16, the emitter base leakage current of transistor T1, and by assuming $R2 = 100k$ the model component values can be calculated using:

1. $I_{S2} = I_{S1} \cdot exp\left(\frac{VOS}{Vt}\right) \cong I_{S1}\left[1 + \frac{VOS}{Vt}\right]$,where Vt = 26e-3 V.
2. $I_{C1} = \frac{C_2 SR^+}{2}$, where SR^+ is the positive slew rate.
3. $I_{C2} = I_{C1}$
4. $I_{B1} = I_B - \frac{I_{OS}}{2}$ and $I_{B2} = I_B + \frac{I_{OS}}{2}$

[14]The use of the SPICE preprocessors SPICEPP and SPICEPRM are described in Qucs tutorial *Qucs simulation of SPICE netlists*. Since both SPICEPP and SPICEPRM were first written, Friedrch Schmidt has developed a PSpice to SPICE3/XSPICE preprocessor which combines, and extends, the features found in both SPICEPP and SPICEPRM. This preprocessor is called PS2SP. The Perl script version of PS2SP is licensed under GPL and may be downloaded from http://members.aon.at/fschmid7/.

[15]See the tutorial *Qucs simulation of SPICE netlist* for instructions on how this can be done.

[16]G.R. Boyle, B.M. Cohn, D. Pederson, and J.E. Solomon, Macromodelling of integrated circuit operational amplifiers, IEEE Journal of Solid State Circuits, vol. SC-9, pp. 353-364, 1974.

[17]See Fig. 7.3. Tests show that the Boyle macromodel reduces simulation times for common amplifier, timer and filter circuits by a factor between six and ten.

5. $B_1 = \dfrac{I_{C1}}{I_{B1}}$ and $B_2 = \dfrac{I_{C2}}{I_{B2}}$
6. $IEE = \left[\dfrac{B_1+1}{B_1} + \dfrac{B_2+1}{B_2}\right] I_{C1}$
7. $RC1 = \dfrac{1}{2\pi GBPC_2}$
8. $RC2 = RC1$
9. $RE1 = \dfrac{B_1+B_2}{2+B_1+B_2}\left[RC1 - \dfrac{1}{gm1}\right]$, where $gm1 = \dfrac{I_{C1}}{Vt}$, and $RE2 = RE1$
10. $CEE = \dfrac{C2}{2} \cdot tan\left(\triangle\phi \dfrac{\pi}{180}\right)$, where $\triangle\phi = 90^o - \Phi m$ and Φm is the phase margin.
11. $GCM = \dfrac{1}{CMMRRC1}$
12. $GA = \dfrac{1}{RC1}$
13. $GB = \dfrac{AvOLRC1}{R2RO2}$
14. $ISD1 = IX \cdot exp\,(TMP1)$+1e-32, where $IX = 2 \cdot I_{C1} \cdot R2 \cdot GB - I_{S1}$, and $TMP1 = \dfrac{-1}{RO1\dfrac{I_{S1}}{Vt}}$
15. $RC = \dfrac{Vt}{100 \cdot IX} ln\,(TEMP2)$, where $TEMP2 = \dfrac{IX}{ISD1}$
16. $VC = abs\,(VCC) - VOUT_P + Vt \cdot ln\left(\dfrac{ISC_P}{I_{S1}}\right)$
17. $VE = abs\,(VEE) + VOUT_N + VT \cdot ln\left(\dfrac{ISC_N}{I_{S1}}\right)$
18. $RP = \dfrac{(VCC - VEE)\,(VCC - VEE)}{PD}$

Rather than calculate the Boyle macromodel component values by hand using a calculator it is better to use a PS2SP preprocessor template that does these calculations and also generates the Boyle SPICE netlist. A template for this task is given in Fig. 7.45. The parameters at the beginning of the listing are for the UA741 OP AMP. In Fig. 7.45 the macromodel internal nodes are indicated by numbers and external nodes by descriptive names. This makes it easier to attach the macromodel interface nodes to a Qucs schematic symbol. The SPICE netlist shown in Fig. 7.46 was generated by SP2SP.

```
*subcircuit ports: in+   in-   p_out  p_vcc p_vee
.subckt opamp_ac in_p in_n p_out p_vcc p_vee
* OP27 OP AMP parameters
.param voff = 30.0u  ib = 15n  ioff = 12n
.param rd = 4meg  cd = 1.4p  cmrrdc = 1.778e6
.param fcmz = 2000.0 aoldc = 1.778e6  gbp = 8meg
.param fp2 = 17meg  pslewr=2.8e6  nslewr=2.8e6
.param vccm=15  vpoutm=14  veem=-15
.param vnoutm=-14  idcoutm=32m  ro=70.0
.param p1={(100*pslewr)/(2*3.1412*gbp) -0.7}
.param p2={(100*nslewr)/(2*3.1412*gbp) -0.7}
* input stage
voff1 in_n 6 {voff/2}
voff2 7 in_p {voff/2}
ib1   0 6 {ib}
ib2   7 0 {ib}
ioff1 7 6 {ioff/2}
r1    6 8 {rd/2}
r2    7 8 {rd/2}
cin1  6 7 {cd}
* common-mode zero stage
ecm1 12 0 8 0 {1e6/cmrrdc}
rcm1 12 13 1meg
ccm1 12 13 {1/(2*3.1412*1e6*fcmz)}
rcm2 13 0 1
* differential and common-mode signal summing stage
gmsum1 0 14 7 6 1
gmsum2 0 14 13 0 1
rsum1  14 0 1
* slew rate stage
gsrc1 0 15 13 0 1
rscale1 15 0 100
dsl 15 16 {dslewrate}
.model dslewrate d(is=1e-12 bv= { p1+p2 } )
vsr1 16 0 {p1}
gmsrt1 0 17 15 0 0.01
rsrt1 17 0 1
* voltage gain stage 1
gmp1 0 9 17 0 1
rado 9 0 {aoldc}
cp1 9 0 {1/(2*3.1412*gbp)}
* voltage gain stage 2
gmp2 0 11 9 0 1
rp2  11 0 1
cp2  11 0 {1/(2*3.1412*fp2)}
* output stage
eos1 10 0 11 0 1
ros1 10 50 {ro}
*output current limiter stage
rdcl1 50 0 100meg
dcl1 21 50 dclim
dcl2 50 21 dclim
.model dclim d(is=1e-15 cj0=0.0)
vcl1 50 p_out 0v
hcl1 0 22 vcl1 {0.9/idcoutm}
ecl1 21 22 50 0 1
* voltage limiting stage
dvl1 p_out 30 dvlimit
.model dvlimit d(is=8e-16)
dvl2 40 p_out dvlimit
vlim1 p_vcc 30 {vcc-vccm+1}
vlim2 40 p_vee {-vee +veem+1}
.ends
.end
```

Figure 7.42: PS2SP template for the OP27 modular macromodel.

```
*subcircuit ports: in+ in- p_out p_vcc p_vee
* infile=op27.pp date=Tue Feb 13 17:32:37 2007 Converted with ps2sp.pl V4.11
* options: -sp3=0 -ltspice=0 -fromsub=0 -fromlib=0 -check=0 (tinylines=1)
* copyright 2007 by Friedrich Schmidt - terms of Gnu Licence
.subckt opamp_ac in_p in_n p_out p_vcc p_vee
voff1 in_n 6 1.5e-05
voff2 7 in_p 1.5e-05
ib1 0 6 1.5e-08
ib2 7 0 1.5e-08
ioff1 7 6 6e-09
r1 6 8 2000000
r2 7 8 2000000
cin1 6 7 1.4e-12
ecm1 12 0 8 0 0.562429696287964
rcm1 12 13 1meg
ccm1 12 13 7.95874188208328e-11
rcm2 13 0 1
gmsum1 0 14 7 6 1
gmsum2 0 14 13 0 1
rsum1 14 0 1
gsrc1 0 15 13 0 1
rscale1 15 0 100
dsl 15 16 0
.model dslewrate d(is=1e-12 bv= 9.7422386349166 )
vsr1 16 0 4.8711193174583
gmsrt1 0 17 15 0 0.01
rsrt1 17 0 1
gmp1 0 9 17 0 1
rado 9 0 1778000
cp1 9 0 1.98968547052082e-08
gmp2 0 11 9 0 1
rp2 11 0 1
cp2 11 0 9.36322574362739e-09
eos1 10 0 11 0 1
ros1 10 50 70
rdcl1 50 0 100meg
dcl1 21 50 dclim
dcl2 50 21 dclim
.model dclim d(is=1e-15 cj0=0.0)
vcl1 50 p_out 0v
hcl1 0 22 vcl1 28.125
ecl1 21 22 50 0 1
dvl1 p_out 30 dvlimit
.model dvlimit d(is=8e-16)
dvl2 40 p_out dvlimit
vlim1 p_vcc 30 -14
vlim2 40 p_vee -14
.ends
.end
```

Figure 7.43: SPICE netlist for the OP27 modular macromodel.

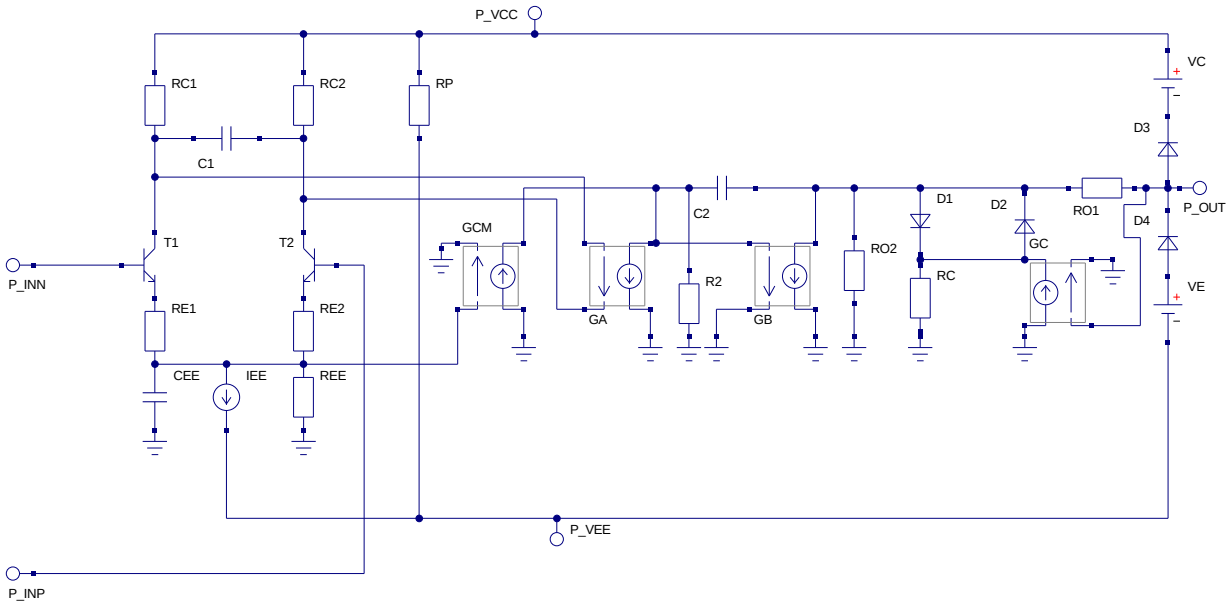


Figure 7.44: Boyle macromodel for a BJT OP AMP

7.11.4 Model accuracy

The modular and Boyle OP AMP macromodels are examples of typical device models in common use with todays popular circuit simulators. A question which often crops up is which model is best to use when simulating a particular circuit? This is a complex question which requires careful consideration. One rule of thumb worth following is **always validate a SPICE/Qucs model before use**. Users can then check that a specific model does simulate the circuit parameters that control the function and accuracy of the circuit being designed[18]. One way to check the performance of a given model is to simulate a specific device parameter. The simulation results can then be compared to manufacturers published figures and the accuracy of a model easily determined. By way of an example consider the simulation circuit shown in Fig. 7.47. In this circuit the capacitors and inductors ensure that the devices under test are in ac open loop mode with stable dc conditions. Figure 7.48 illustrates the observed simulation gain and phase results for four different OP AMP models. Except at very high frequencies, which are outside device normal operating range, good agreement is found between manufacturers data and that recorded by the open loop voltage gain test for both the modular and Boyle macromodels.

[18] An interesting series of articles by Ron Mancini, on verification and use of SPICE models in circuit design can be found in the following editions of EDN magazine:Validate SPICE models before use, EDN March 31, 2005 p.22; Understanding SPICE models, EDN April 14, p 32; Verify your ac SPICE model, EDN May 26, 2005; Beyond the SPICE model's dc and ac performance, EDN June 23 2005, and Compare SPICE-model performance, EDN August 18, 2005.

```
* Boyle macromodel template for Qucs.
* Design parameters (For UA741)
.param vt=26e-3  $ Thermal voltage at room temp.
.param c2=30e-12    $ Compensation capacitance
.param positive_slew_rate=0.625e6 negative_slew_rate=0.50e6  $ Slew rates
.param is1=8.0e-16  $ T1 leakage current
.param vos=0.7e-3 ib=80n ios=20n  $ Input voltage and current parameters
.param va=200  $ Nominal early voltage
.param gbp=1.0e6 $ Gain bandwidth product
.param pm=70 $ Excess phase at unity gain.
.param cmrr=31622.8 $ Common-mode rejection ratio (90 dB)
.param avol=200k $ DC open loop differential gain
.param ro2=489.2 $ DC output resistance
.param ro1=76.8 $ High frequency AC output resistance
.param r2=100k
.param vout_p=14.2 $ Positive saturation voltage - for VCC=15v
.param vout_n=-13.5 $ Negative saturation voltage - for VCC=-15v
.param vcc=15 $ Positive power supply voltage
.param vee=-15 $ Negative power supply voltage
.param isc_p=25m $ Short circuit output current
.param isc_n=25m $ Short circuit output current
.param pd=59.4m $ Typical power dissipation
* Design equations
.param is2={is1*(1+vos/vt)}
.param ic1={0.5*c2*positive_slew_rate} ic2={ic1}
.param ib1={ib-0.5*ios} ib2={ib+0.5*ios}
.param b1={ic1/ib1} b2={ic2/ib2}
.param iee={((b1+1)/b1+(b2+1)/b2)*ic1}
.param gm1={ic1/vt} rc1={1/(2*3.1412*gbp*c2)} rc2=rc1
.param re1={((b1+b2)/(2+b1+b2))*(rc1-1/gm1)} re2=re1
.param ree={va/iee} cee={(2*ic1/negative_slew_rate)-c2}
.param dphi={90-pm} c1={(c2/2)*tan(dphi*3.1412/180)}
.param gcm={1/(cmrr*rc1)} ga={1/rc1} gb={(avol*rc1)/(r2*ro2)}
.param ix={2*ic1*r2*gb-is1} tmp1={-1.0/(ro1*is1/vt)} isd1={ix*exp(tmp1)+1e-32}
.param tmp2={ix/isd1} rc={vt/(100*ix)*ln(tmp2)}
.param gc={1/rc}
.param vc={abs(vcc)-vout_p+vt*ln(isc_p/is1)} ve={abs(vee)+vout_n+vt*ln(isc_n/is1)}
.param rp={(vcc-vee)*(vcc-vee)/pd}
* Nodes: Input n_inp n_inn n_vcc n_vee   Output n_out
Q1 8 n_inn 10 qmod1
Q2 9 n_inp 11 qmod2
RC1 n_vcc 8 {rc1}
RC2 n_vcc 9 {rc2}
RE1 1 10 {re1}
RE2 1 11 {re2}
RE 1 0 {ree}
CE 1 0 {cee}
IEE 1 n_vee {iee}
C1 8 9 {c1}
RP n_vcc n_vee {rp}
GCM 0 12 1 0 {gcm}
GA 12 0 8 9 {ga}
R2 12 0 {r2}
C2 12 13 30p
GB 13 0 12 0 {gb}
RO2 13 0 {ro2}
RO1 13 n_out {ro1}
D1 13 14 dmod1
D2 14 13 dmod1
GC 0 14 n_out 0 {gc}
RC 14 0 {rc}
D3 n_out 15 DMOD3
D4 16 n_out DMOD3
VC n_vcc 15 {vc}
VE 16 n_vee {ve}
.model dmod1 d(is={isd1} rs=1)
.model dmod3 d(is=8e-16 rs=1)
.model qmod1 npn(is={is1} BF={b1})
.model qmod2 npn(is={is2} BF={b2})
.end
```


Figure 7.45: PS2SP template for the Boyle macromodel with UA741 parameters listed.

```
* boyle macromodel template for qucs.
* infile=ua741_boyle.ps2sp date=Tue Feb 6 20:58:12 2007 Converted with ps2sp.pl V4.11
* options: -sp3=0 -ltspice=0 -fromsub=0 -fromlib=0 -check=0 (tinylines=1)
* copyright 2007 by Friedrich Schmidt - terms of Gnu Licence
q1 8 n_inn 10 qmod1
q2 9 n_inp 11 qmod2
rc1 n_vcc 8 5305.82792138885
rc2 n_vcc 9 5305.82792138885
re1 1 10 1820.05072213971
re2 1 11 1820.05072213971
re 1 0 13192612.1372032
ce 1 0 7.5e-12
iee 1 n_vee 1.516e-05
c1 8 9 5.4588124089082e-12
rp n_vcc n_vee 15151.5151515152
gcm 0 12 1 0 5.96000354174836e-09
ga 12 0 8 9 0.000188472
r2 12 0 100000
c2 12 13 30p
gb 13 0 12 0 21.6918557701915
ro2 13 0 489.2
ro1 13 n_out 76.8
d1 13 14 dmod1
d2 14 13 dmod1
gc 0 14 n_out 0 1621.78603105575
rc 14 0 0.000616604151750539
d3 n_out 15 dmod3
d4 16 n_out dmod3
vc n_vcc 15 1.60789905279489
ve 16 n_vee 2.30789905279488
.model dmod1 d(is=1e-32 rs=1)
.model dmod3 d(is=8e-16 rs=1)
.model qmod1 npn(is=8e-16 bf=107.142857142857)
.model qmod2 npn(is=8.215384615384 61e-16 bf=83.3333333333333)
.end
```

Figure 7.46: SPICE netlist for the Boyle UA741 macromodel.

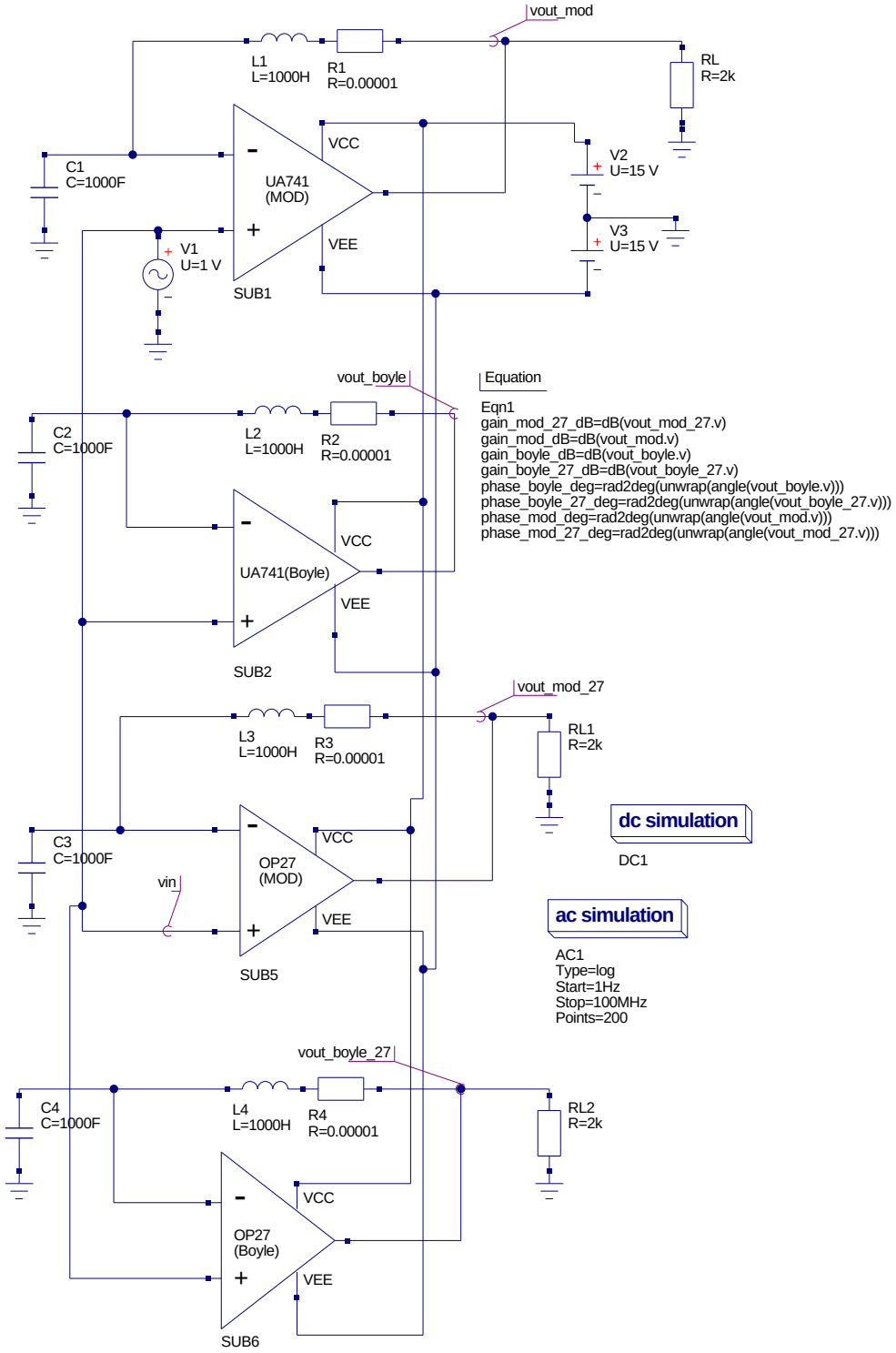


Figure 7.47: Test circuit for simulating OP AMP model open loop voltage gain.

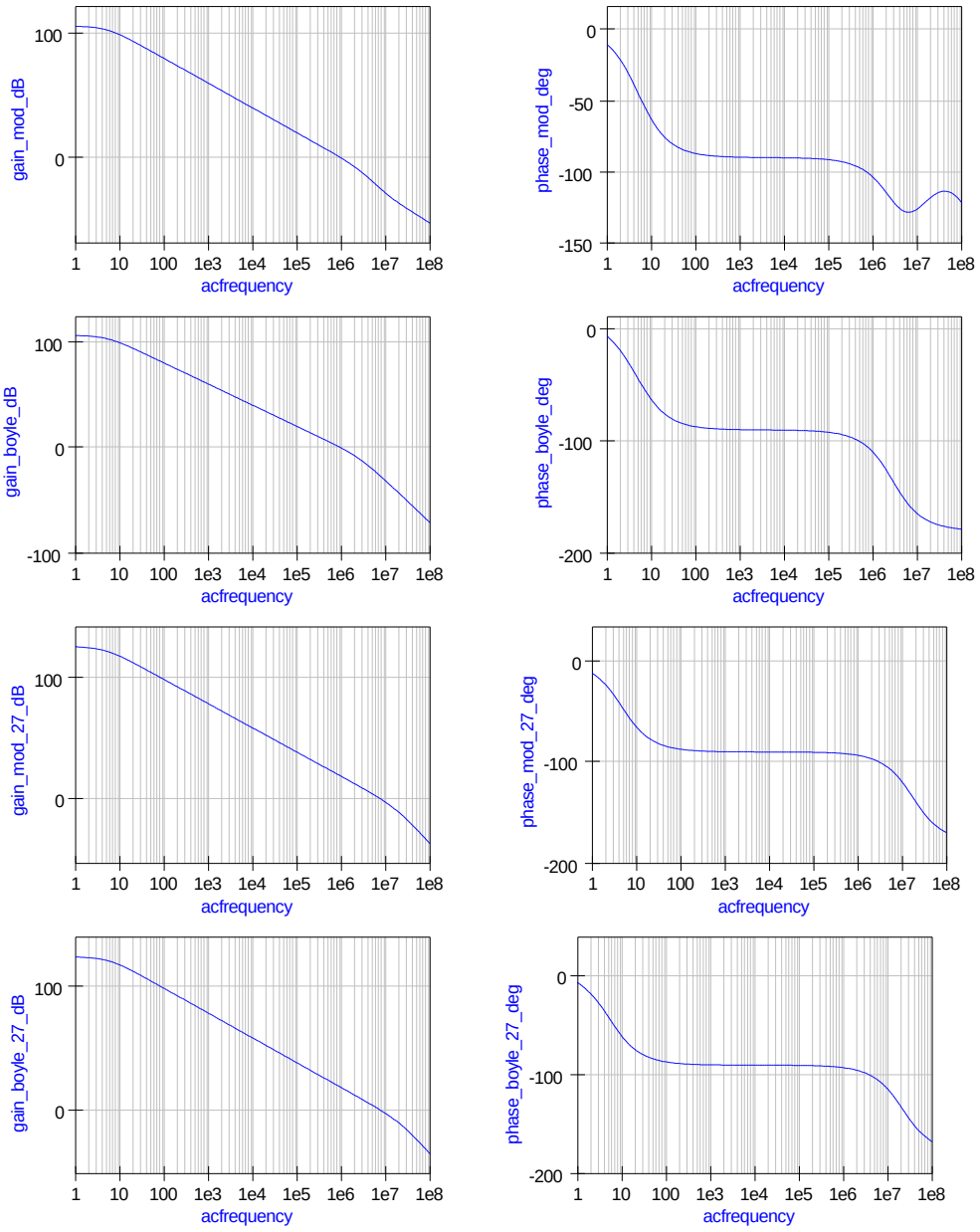


Figure 7.48: Open loop voltage gain simulation waveforms for the modular and Boyle UA741 and OP27 macromodels.

7.11.5 The PSpice modified Boyle model

One of the most widely used OP AMP simulation models is a modified version of the Boyle macromodel. This was originally developed for use with the PSpice circuit simulator. Many semiconductor manufacturers provide models for their devices based on the modified Boyle macromodel[19]. A typical modified Boyle macromodel SPICE netlist is shown in Fig. 7.49. The circuit structure and performance are very similar, but significantly different to the original Boyle model. Some of the common OP AMP parameters NOT modeled are (1) input offset voltage, (2) temperature coefficient of input offset voltage, (3) input offset current, (4) equivalent input voltage and noise currents, (5) common-mode input voltage range, and (6) temperature effect on component stability. Items (2), (4), (5) and (6) are also not modeled by the standard Boyle macromodel. Although the modified Boyle macromodel is similar to the original Boyle model it is not possible to use this model as it is defined with Qucs; due to the fact that SPICE 2G nonlinear controlled sources, egnd and fb, are included in the SPICE netlist. Controlled source egnd is employed to model the OP AMP reference voltage as the average of the VCC and VEE power rail voltages rather than the ground voltage assumed in the original Boyle macromodel[20]. Current conrolled current source fb is used to model OP AMP output current limiting. The nonlinear polynomial[21] form of controlled sources were included in the 2G series of SPICE simulators to allow behavioural models of summers, multipliers, buffers and other important functional components to be easily constructed. Single and multidimensional polynomial forms of controlled sources are defined by SPICE 2G. Taking (1) the voltage controlled voltage source and (2) the current controlled current sources as examples the syntax is as follows:

`Ename N(+) N(-) POLY(n) NC1(+) NC1(-) NC2(+) NC2(-)..... P0  P1  P2.....,`
where n indicates the order of the polynomial with coefficients P0 .....Pn, and NCn(+), NCn(-) etc are the control node pairs.
This becomes:

- For POLY(1) [22]: `Ename N(+) N(-) P0 P1 P2.........`
- For POLY(2): `Ename N(+) N(-) POLY(2) NC1(+) NC1(-) NC2(+) NC(-) P0 P1 P2......`
- For POLY(3): `Ename N(+) N(-) POLY(3) NC1(+) NC1(-) NC2(+) NC2(-) NC3(+) NC3(-) P0 P1 P2....`
  and so on.

[19] See for example the OP AMP section of the Texas Instruments (TI) Web site and the TI Operational Amplifier Circuits, Linear Circuits, Data Manual, 1990.

[20] Taking the OP AMP reference voltage to be the average of VCC and VEE allows devices with non-symmetrical power supply voltages to be simulated.

[21] The definition of these polynomial functions was changed in the SPICE 3 series simulators to a more conventional algebraic form when specifying the B type source components. This often gives compatibility problems when attempting to simulate SPICE 2 models with circuit simulators developed from SPICE 3f4 or earlier simulators. Most popular SPICE based circuit simulators now accept both types of nonlinear syntax.

[22] If only one P coefficient is given in the single dimension polynomial case, then SPICE assumes that this is P1 and that P0 equals zero. Similarly if the POLY keyword is not explicitly stated in a controlled source definition then it is assumed by SPICE to be POLY(1).

Similarly: Fname N(+) N(-) POLY(n) V1 V2 V3 ..... P0 P1 P2 ....., where V1, V2 .... are independent voltage sources whose current controls the output. This becomes:

- For POLY(1): Fname N(+) N(-) V1 P0 P1 P2.....
- For POLY(2): Fname N(+) N(-) POLY(2) V1 V2 P0 P1 P2.....
- For POLY(3): Fname N(+) N(-) POLY(3) V1 V2 V3 P0 P1 P2 P3....., and so on.

The meaning of the coefficients in the nonlinear controlled source definitions depends on the dimension of the polynomial. The following examples indicate how SPICE calculates current or voltage values.

- For POLY(1): The polynomial function fv is calculated using $fv = P0 + (P1 * fa) + (P2 * fa^2) + (P3 * fa^3) + (P4 * fa^4) +$ ....., where fa is either a voltage or current independent variable.
- For POLY(2): The polynomial function fv is calculated using $fv = P0 + (P1 * fa) + (P2 * fb) + (P3 * fa^2) + (P4 * fa * fb) + (P5 * fb^2) + (P6 * fa^3)$ $+(P7 * fa^2 * fb) +$ ....., where fa and fb are both either voltage or current independent variables.
- For POLY(3): The polynomial function fv is calculated using $fv = P0 + (P1 * fa) + (P2 * fb) + (P3 * fc) + (P4 * fa^2) + (P5 * fa * fb)$ $+(P6 * fa * fc) + (P7 * fb^2) + (P8 * fb * fc) + (P9 * fc^2) + (P10 * fa^3) +(P11 * fa^2 * fb) + (P12 * fa^2 * fc) + (P13 * fa * fb^2) + (P14 * fa * fb * fc) +(P15 * fa * fc^2) + (P16 * fb^3) + (P17 * fb^2 * fc) + (P18 * fb * fc^2) +(P19 * fc^3)$....., where fa, fb, and fc are all either voltage or current independent variables.

From Fig. 7.49 the controlled generators egnd and fb are:

- `egnd 99  0 poly(2) (3,0) (4,0) 0 .5 .5`
  Which is the same as `egnd 99 0 poly(2) 3 0 4 0 0 0.5 0.5`
  By comparison with the SPICE polynomial equations for controlled sources, $V(egnd) = \frac{V(3)}{2} + \frac{V(4)}{2}$ implying that the controlled voltage source $V(egnd)$ is the sum of two linear voltage sources.
- `fb 7 99 poly(5) vb vc ve vlp vln 0 10.61E6 -10E6 10E6 10E6 -10E6`
  By comparison with the SPICE polynomial equations for controlled sources `I(fb) = 10.61e6*I(vb) - 10e6*I(vc) + 10e6*I(ve) + 10e6*I(vlp) -10e6*I(vlp)` implying that the controlled current $I(fb)$ is the sum of five linear controlled current sources.

SPICE sources engd and fb can therefore be replaced in the modified Boyle model by the following SPICE code[23]:

```
* egnd 99  0 poly(2) (3,0) (4,0) 0 .5 .5
* Forms voltage source with output
* V=0.5*V(4)+0.5*V(3)
egnd1 999 0 4 0 0.5
egnd2 99 999 3 0 0.5
*
*fb 7 99 poly(5) vb vc ve vlp vln 0 10.61e6 -10e6 10e6 10e6 -10e6
*
* Forms current source with output
* I=10.61e6*i(vb)-10e6*i(vc)+10e6*i(ve)+10e6*i(vlp)-10e6*i(vln)
*
*Sum 5 current sources to give fb.
fb1 7 99 vb 10.61e6
fb2 7 99 vc -10e6
fb3 7 99 ve 10e6
fb4 7 99 vlp 10e6
fb5 7 99 vln -10e6
```

Modified Boyle macromodels are often generated using the PSpice Parts[24] program. Such models have similar structured SPICE netlists with different component values. However, changes in technology do result in changes in the input stage that reflect the use of npn, pnp and JFET input transistors in real OP AMPs. Hence to use manufacturers published modified Boyle models with Qucs all that is required is the replacement of the SPICE polynomial controlled sources with linear sources and the correct component values. Again this is best done using a SPICE preprocessor template. The templates for OP AMPS with npn and PJF input transistors are shown in Figures 7.50 and 7.51. The SPICE netlists shown in Figs. 7.52 and 7.53 were generated by the PS2SP preprocessor. For OP AMPS with pnp input transistors simply change the BJT model reference from npn to pnp and use the same template.

[23]It is worth noting that the code for the polynomial form of controlled sources can only be replaced by a series connection of linear controlled voltage sources or a parallel connection of linear controlled current sources provided no higher order polynomial coefficients are present in the original SPICE code. Some SPICE models use these higher order coefficients to generate multiply functions. Such cases cannot be converted to code which will simulate using Qucs 0.0.10. Sometime in the future this restriction will be removed when nonlinear voltage and current sources are added to Qucs.

[24]The Parts modelling program is an integral component in the PSpice circuit simulation software originally developed by the MicroSim Corporation, 1993, The Design Centre:Parts (Irvine, Calif.). It now forms part of Cadence Design Systems OrCad suite of CAD software.

```
* connections:   non-inverting input
*                |  inverting input
*                |  |  positive power supply
*                |  |  |  negative power supply
*                |  |  |  |  output
*                |  |  |  |  |
.subckt uA741     1  2  3  4  5
*
  c1   11 12 8.661E-12
  c2    6  7 30.00E-12
  dc    5 53 dx
  de   54  5 dx
  dlp  90 91 dx
  dln  92 90 dx
  dp    4  3 dx
  egnd 99  0 poly(2) (3,0) (4,0) 0 .5 .5
  fb    7 99 poly(5) vb vc ve vlp vln 0 10.61E6 -10E6 10E6 10E6 -10E6
  ga    6  0 11 12 188.5E-6
  gcm   0  6 10 99 5.961E-9
  iee  10  4 dc 15.16E-6
  hlim 90  0 vlim 1K
  q1   11  2 13 qx
  q2   12  1 14 qx
  r2    6  9 100.0E3
  rc1   3 11 5.305E3
  rc2   3 12 5.305E3
  re1  13 10 1.836E3
  re2  14 10 1.836E3
  ree  10 99 13.19E6
  ro1   8  5 50
  ro2   7 99 100
  rp    3  4 18.16E3
  vb    9  0 dc 0
  vc    3 53 dc 1
  ve   54  4 dc 1
  vlim  7  8 dc 0
  vlp  91  0 dc 40
  vln   0 92 dc 40
.model dx D(Is=800.0E-18 Rs=1)
.model qx NPN(Is=800.0E-18 Bf=93.75)
.ends
```

Figure 7.49: PSpice modified Boyle macromodel for the UA741 OP AMP.

```
* Modified Boyle OP AMP model template
* npn BJT input devices.
*
*UA741C OP AMP parameters, manufacturer Texas Instruments
.param c1=4.664p c2=20.0p
.param ep1=0.5 ep2=0.5
.param fp1=10.61e6 fp2=-10e6 fp3=10e6 fp4=10e6 fp5=-10e6
.param vc=2.6 ve=2.6 vlp=25 vln=25
.param ga=137.7e-6 gcm=2.57e-9
.param iee=10.16e-6 hlim=1k
.param r2=100k
.param rc1=7.957k rc2=7.957k
.param re1=2.74k re2=2.74k
.param ree=19.69e6 ro1=150 ro2=150
.param rp=18.11k
*
.subckt ua741_TI P_INP P_INN P_VCC P_VEE P_OUT
c1 11 12 {c1}
c2 6 7 {c2}
dc P_OUT 53 dx
de 54 P_OUT dx
dlp 90 91 dx
dln 92 90 dx
* egnd 99 0 poly(2) (3,0) (4,0) 0 0.5 0.5
* Qucs modification, Mike Brinson, Feb 2007
egnd1 999 0 P_VCC 0 {ep1}
egnd2 99 999 P_VEE 0 {ep2}
*fb 7 99 poly(5) vb vc ve vlp vln 0 10.61e6 -10e6 10e6 10e6 -10e6
* Forms current source with output
* I=10.61e6*i(vb)-10e6*i(vc)+10e6*i(ve)+10e6*i(vlp)-10e6*i(vln)
*Qucs modification, Mike Brinson, Feb 2007.
*Sum 5 current sources to give fb.
fb1 7 99 vb {fp1}
fb2 7 99 vc {fp2}
fb3 7 99 ve {fp3}
fb4 7 99 vlp {fp4}
fb5 7 99 vln {fp5}
*
ga 6 0 11 12 {ga}
gcm 0 6 10 99 {gcm}
iee 10 P_VEE {iee}
hlim 90 0 vlim {hlim}
q1 11 P_INN 13 qx
q2 12 P_INP 14 qx
r2 6 9 100k
rc1 P_VCC 11 {rc1}
rc2 P_VCC 12 {rc2}
re1 13 10 {re1}
re2 14 10 {re2}
ree 10 99 {ree}
ro1 8 P_OUT {ro1}
ro2 7 99 {ro2}
rp P_VCC P_VEE {rp}
vb 9 0 dc 0
vc P_VCC 53 dc {vc}
ve 54 P_VEE dc {ve}
vlim 7 8 dc 0
vlp 91 0 dc {vlp}
vln 0 92 dc {vlp}
.model dx d(is=800.0e-18)
.model qx npn(is=800.0e-18 bf=62.5)
.ends
.end
```

Figure 7.50: Modified Boyle PS2SP netlist for the UA741 OP AMP.

```
* Modified Boyle OP AMP model template
* JFET input devices.
*
*TL081 OP AMP parameters, manufacturer Texas Instruments
.param c1=3.498p c2=15.0p
.param ep1=0.5 ep2=0.5
.param fp1=4.715e6 fp2=-5e6 fp3=5e6 fp4=5e6 fp5=-5e6
.param vc=2.2 ve=2.2 vlp=25 vln=25
.param ga=282.8e-6 gcm=8.942e-9
.param iss=195.0e-6 hlim=1k
.param r2=100k
.param rd1=3.536k rd2=3.536k
.param rss=1.026e6 ro1=150 ro2=150
.param rp=2.14k
*
.subckt ua741_TI P_INP P_INN P_VCC P_VEE P_OUT
c1 11 12 {c1}
c2 6 7 {c2}
dc P_OUT 53 dx
de 54 P_OUT dx
dlp 90 91 dx
dln 92 90 dx
* egnd 99 0 poly(2) (3,0) (4,0) 0 0.5 0.5
* Qucs modification, Mike Brinson, Feb 2007
egnd1 999 0 P_VCC 0 {ep1}
egnd2 99 999 P_VEE 0 {ep2}
*fb 7 99 poly(5) vb vc ve vlp vln 0 10.61e6 -10e6 10e6 10e6 -10e6
* Forms current source with output
* I=10.61e6*i(vb)-10e6*i(vc)+10e6*i(ve)+10e6*i(vlp)-10e6*i(vln)
*Qucs modification, Mike Brinson, Feb 2007.
*Sum 5 current sources to give fb.
fb1 7 99 vb {fp1}
fb2 7 99 vc {fp2}
fb3 7 99 ve {fp3}
fb4 7 99 vlp {fp4}
fb5 7 99 vln {fp5}
*
ga 6 0 11 12 {ga}
gcm 0 6 10 99 {gcm}
iss P_VCC 10 {iss}
hlim 90 0 vlim {hlim}
j1 11 P_INN 10 jx
j2 12 P_INP 10 jx
r2 6 9 100k
rd1 P_VEE 11 {rd1}
rd2 P_VEE 12 {rd2}
ro1 8 P_OUT {ro1}
ro2 7 99 {ro2}
rp P_VCC P_VEE {rp}
rss 10 99 {rss}
vb 9 0 dc 0
vc P_VCC 53 dc {vc}
ve 54 P_VEE dc {ve}
vlim 7 8 dc 0
vlp 91 0 dc {vlp}
vln 0 92 dc {vlp}
.model dx d(is=800.0e-18)
.model jx pjf(is=15.0e-12 beta=270.1e-6 vto=-1)
.ends
.end
```

Figure 7.51: Modified Boyle PS2SP netlist for the TL081 OP AMP.

```
* modified boyle op amp model template
* infile=Mod_boyle_template_npn.pp date=Thu Feb 8 23:54:59 2007 Converted with ps2sp.pl V4.11
* options: -sp3=1 -ltspice=0 -fromsub=0 -fromlib=0 -check=0 (tinylines=1)
* copyright 2007 by Friedrich Schmidt - terms of Gnu Licence
.subckt ua741_ti p_inp p_inn p_vcc p_vee p_out
c1 11 12 4.664e-12
c2 6 7 2e-11
dc p_out 53 dx
de 54 p_out dx
dlp 90 91 dx
dln 92 90 dx
egnd1 999 0 p_vcc 0 0.5
egnd2 99 999 p_vee 0 0.5
fb1 7 99 vb 9.42507068803016e-08
fb2 7 99 vc -1e-07
fb3 7 99 ve 1e-07
fb4 7 99 vlp 1e-07
fb5 7 99 vln -1e-07
ga 6 0 11 12 0.0001377
gcm 0 6 10 99 2.57e-09
iee 10 p_vee 1.016e-05
hlim 90 0 vlim 0.001
q1 11 p_inn 13 qx
q2 12 p_inp 14 qx
r2 6 9 100k
rc1 p_vcc 11 7957
rc2 p_vcc 12 7957
re1 13 10 2740
re2 14 10 2740
ree 10 99 19690000
ro1 8 p_out 150
ro2 7 99 150
rp p_vcc p_vee 18110
vb 9 0 0
vc p_vcc 53 2.6
ve 54 p_vee 2.6
vlim 7 8 0
vlp 91 0 25
vln 0 92 25
.model dx d(is=800.0e-18)
.model qx npn(is=800.0e-18 bf=62.5)
.ends
.end
```

Figure 7.52: Modified Boyle SPICE netlist for the TI UA741 OP AMP.

```
* modified boyle op amp model template
* infile=TL081_TI.pp date=Sun Feb 11 16:04:22 2007 Converted with ps2sp.pl V4.11
* options: -sp3=0 -ltspice=0 -fromsub=0 -fromlib=0 -check=0 (tinylines=1)
* copyright 2007 by Friedrich Schmidt - terms of Gnu Licence
.subckt ua741_ti p_inp p_inn p_vcc p_vee p_out times
c1 11 12 3.498e-12
c2 6 7 1.5e-11
dc p_out 53 dx
de 54 p_out dx
dlp 90 91 dx
dln 92 90 dx
egnd1 999 0 p_vcc 0 0.5
egnd2 99 999 p_vee 0 0.5
fb1 7 99 vb 4715000
fb2 7 99 vc -5000000
fb3 7 99 ve 5000000
fb4 7 99 vlp 5000000
fb5 7 99 vln -5000000
ga 6 0 11 12 0.0002828
gcm 0 6 10 99 8.942e-09
iss p_vcc 10 0.000195
hlim 90 0 vlim 1000
j1 11 p_inn 10 jx
j2 12 p_inp 10 jx
r2 6 9 100k
rd1 p_vee 11 3536
rd2 p_vee 12 3536
ro1 8 p_out 150
ro2 7 99 150
rp p_vcc p_vee 2140
rss 10 99 1026000
vb 9 0 dc 0
vc p_vcc 53 dc 2.2
ve 54 p_vee dc 2.2
vlim 7 8 dc 0
vlp 91 0 dc 25
vln 0 92 dc 25
.model dx d(is=800.0e-18)
.model jx pjf(is=15.0e-12 beta=270.1e-6 vto=-1)
.ends
.end
```

Figure 7.53: Modified Boyle SPICE netlist for the TI TL081 OP AMP.

7.12 Constructing Qucs OPAMP libraries

Qucs release 0.0.10 includes a facility which allows users to build their own component libraries. This facility can be used to construct any library which contains device models formed using the standard schematic entry route provided the individual components that make up a model do not contain components that require file netlists. Qucs, for example, converts SPICE netlists to Qucs formated netlists when a simulation is performed but does not retain the converted netlists. Hence, to add OP AMP macromodels that are based on SPICE netlist to a Qucs library a slightly modified procedure is required that involves users copying the converted SPICE netlist into a Qucs library. One way for generating SPICE netlist based OP AMP models is as follows[25]:

1. Construct a Qucs OP AMP model using the procedure described on page 3 of the *Qucs Simulation of SPICE Netlists* tutorial.

2. Add this model to a user defined library using the Qucs Create Library facility (short cut Ctrl+Shift+L).

3. Place a copy of the OP AMP model on a drawing sheet and undertake a DC analysis. NOTE: drag and drop the model symbol of the device you are simulating from your current work project and NOT from the newly created Qucs library.

4. Copy the section of the Qucs netlist that has been converted from the model's SPICE netlist and paste this into the newly created library model. The converted SPICE netlist can be displayed by pressing key F6. User generated library files are held in directory `user_lib`.[26]

To demonstrate the procedure consider the following example based on the UA741 Boyle model:
Steps 1 and 2 result in the following entry in a user created library:

```
<Component ua741(boyle)>
  <Description>
UA741 Boyle macromodel
  </Description>
  <Model>
.Def:Lib_OPAMP_ua741_boyle_ _net0 _net1 _net2 _net3 _net4
Sub:X1 _net0 _net1 _net2 _net3 _net4 gnd Type="ua741_boyle_cir"
.Def:End
  </Model>
  <Symbol>
    <.ID -20 74 SUB>
    <Line -20 60 0 -125 #00007f 2 1>
    <Line -20 -65 100 65 #00007f 2 1>
    <Line -20 60 100 -60 #00007f 2 1>
    <Line -35 -35 15 0 #00007f 2 1>
    <Line -35 40 15 0 #00007f 2 1>
```

[25]The procedure presented here must be considered a work around and may change as Qucs develops.
[26]The location of the user created libraries will differ from system to system depending where .qucs is installed.

```
    <Line 80 0 15 0 #00007f 2 1>
    <.PortSym -35 -35 1 0>
    <.PortSym -35 40 2 0>
    <.PortSym 95 0 3 180>
    <Line 60 50 0 -40 #00007f 2 1>
    <Line 60 -15 0 -40 #00007f 2 1>
    <Text -15 -55 30 #000000 0 "-">
    <Text -15 30 20 #000000 0 "+">
    <Text -15 -5 12 #000000 0 "UA741(Boyle)">
    <.PortSym 60 -55 4 180>
    <.PortSym 60 50 5 180>
    <Text 65 -30 12 #000000 0 "VCC">
    <Text 65 20 12 #000000 0 "VEE">
  </Symbol>
</Component>
```

Note that the model requires a subcircuit of type `ua741_boyle_cir` which is not included when the library is created by Qucs. After completing the cut and paste operation described in steps 3 and 4 above the resulting library entry becomes the Qucs netlist shown next.

```
<Component ua741(boyle)>
  <Description>
UA741 Boyle macromodel
  </Description>
  <Model>
.Def:Lib_OPAMP_ua741_boyle_ _net0 _net1 _net2 _net3 _net4
Sub:X1 _net0 _net1 _net2 _net3 _net4 gnd Type="ua741_boyle_cir"
.Def:End
.Def:ua741_boyle_cir _netN_INN _netN_INP _netN_OUT _netN_VCC _netN_VEE _ref
  Vdc:VE _net16 _netN_VEE U="2.3079"
  Vdc:VC _netN_VCC _net15 U="1.6079"
  Diode:D4 _netN_OUT _net16 Is="8e-16" Rs="1" N="1" M="0.5" Cj0="1e-14" Vj="0.7"
  Diode:D3 _net15 _netN_OUT Is="8e-16" Rs="1" N="1" M="0.5" Cj0="1e-14" Vj="0.7"
  R:RC _net14 _ref R="0.000616604"
  VCCS:GC _netN_OUT _ref _net14 _ref G="1621.79"
  Diode:D2 _net13 _net14 Is="1e-32" Rs="1" N="1" M="0.5" Cj0="1e-14" Vj="0.7"
  Diode:D1 _net14 _net13 Is="1e-32" Rs="1" N="1" M="0.5" Cj0="1e-14" Vj="0.7"
  R:RO1 _net13 _netN_OUT R="76.8"
  R:RO2 _net13 _ref R="489.2"
  VCCS:GB _net12 _net13 _ref _ref G="21.6919"
  C:C2 _net12 _net13 C="30p"
  R:R2 _net12 _ref R="100000"
  VCCS:GA _net8 _net12 _ref _net9 G="0.000188472"
  VCCS:GCM _net1 _ref _net12 _ref G="5.96e-09"
  R:RP _netN_VCC _netN_VEE R="15151.5"
  C:C1 _net8 _net9 C="5.45881e-12"
  Idc:IEE _netN_VEE _net1 I="1.516e-05"
  C:CE _net1 _ref C="0"
  R:RE _net1 _ref R="1.31926e+07"
  R:RE2 _net1 _net11 R="1820.05"
  R:RE1 _net1 _net10 R="1820.05"
  R:RC2 _netN_VCC _net9 R="5305.83"
  R:RC1 _netN_VCC _net8 R="5305.83"
  BJT:Q2 _netN_INP _net9 _net11 _ref Type="npn" Is="8.21538e-16" Bf="83.3333" Nf="1" Nr="1" Ikf="0"
  Ikr="0" Vaf="0" Var="0" Ise="0" Ne="1.5" Isc="0" Nc="2" Br="1" Rbm="0" Irb="0" Cje="0" Vje="0.75"
  Mje="0.33" Cjc="0" Vjc="0.75" Mjc="0.33" Xcjc="1" Cjs="0" Vjs="0.75" Mjs="0" Fc="0.5" Vtf="0"
  Tf="0" Xtf="0" Itf="0" Tr="0"
  BJT:Q1 _netN_INN _net8 _net10 _ref Type="npn" Is="8e-16" Bf="107.143" Nf="1"
  Nr="1" Ikf="0" Ikr="0" Vaf="0" Var="0" Ise="0" Ne="1.5" Isc="0" Nc="2" Br="1"
  Rbm="0" Irb="0" Cje="0" Vje="0.75" Mje="0.33" Cjc="0" Vjc="0.75" Mjc="0.33"
  Xcjc="1" Cjs="0" Vjs="0.75" Mjs="0" Fc="0.5" Vtf="0" Tf="0" Xtf="0" Itf="0" Tr="0"
.Def:End
  </Model>
  <Symbol>
```

```
    <.ID -20 74 SUB>
    <Line -20 60 0 -125 #00007f 2 1>
    <Line -20 -65 100 65 #00007f 2 1>
    <Line -20 60 100 -60 #00007f 2 1>
    <Line -35 -35 15 0 #00007f 2 1>
    <Line -35 40 15 0 #00007f 2 1>
    <Line 80 0 15 0 #00007f 2 1>
    <.PortSym -35 -35 1 0>
    <.PortSym -35 40 2 0>
    <.PortSym 95 0 3 180>
    <Line 60 50 0 -40 #00007f 2 1>
    <Line 60 -15 0 -40 #00007f 2 1>
    <Text -15 -55 30 #000000 0 "-">
    <Text -15 30 20 #000000 0 "+">
    <Text -15 -5 12 #000000 0 "UA741(Boyle)">
    <.PortSym 60 -55 4 180>
    <.PortSym 60 50 5 180>
    <Text 65 -30 12 #000000 0 "VCC">
    <Text 65 20 12 #000000 0 "VEE">
  </Symbol>
</Component>
```

7.13 Extending existing OP AMP models

The modular, Boyle and modified Boyle OP AMP models are three popular macromodels selected from a large number of different models that are in common use today. Most device manufacturers provide similar macromodels, or extended versions which more accurately model the performance of specific devices. Indeed, a growing trend has developed which mixes Boyle type models with modular structures[27]. Often, in practical design projects specific OP AMP properties must be simulated which are not modelled with an available OP AMP model. Two approaches can be used to overcome such deficiencies; firstly, a macromodel itself can be modified so that it models the required additional attributes, or secondly external components can be added which again extend model performance.
One important OP AMP parameter that the standard and modified Boyle models do not model is the frequency dependence of amplifier common-mode gain. Only the dc value of the CMRR is modelled. Such frequency dependency can be added by a simple modification[28], requiring one extra node, that simulates ac CMRR and gives close agreement between macromodel performance and data sheet specifications. Components CEE, REE and GCM, see Fig. 7.44, are replaced by the network shown in Fig. 7.54. Data sheets for the UA741 show the CMRR falling above a break frequency of about 200 Hz, due to the zero generated by CEE causing the common-mode gain to increase. This effect can be simulated in the Boyle macromodel by the addition of one extra node and two extra resistors and changes to REE and controlled source GCM as in Fig. 7.44. In this modified network, the common-mode voltage is detected at the junction of RE4 and CEE, introducing a zero into the response and attenuating the signal. The frequency of the zero is set by

[27] See for example: Ray Kendall, User-friendly model simplifies SPICE OP-AMP simulation, EDN magazine, January 4, 2007, pp. 63-69.

[28] This section is based on unpublished work by David Faulkner and Mike Brinson., Department of Computing, Communications Technology and Mathematical Sciences, London Metropolitan University, UK.

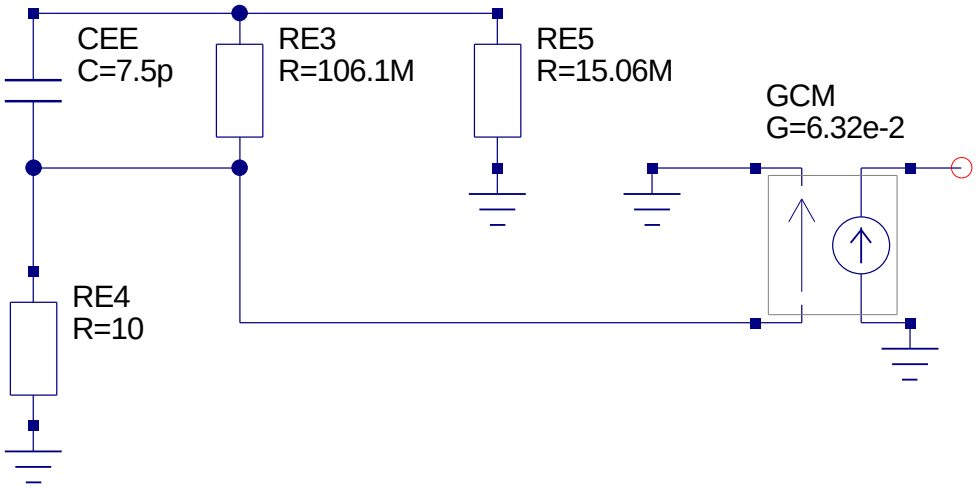


Figure 7.54: AC CMRR modification for Boyle macromodel

$\frac{1}{2*\pi*CEE*RE3}$. The new value of CEE must have the same value as the original CEE value[29] if the same slew-rate is to be maintained, so for a 200 Hz cut-off this gives RE3=106.1M. RE4 is arbitrarily fixed at 10 Ω, which introduces another pole at about 2 GHz, well outside the frequency of interest. The value of REE is increased to RE5 (15.06meg), so that RE5 in parallel with RE3 equals the original value of REE. GCM is also increased by the factor $\frac{RE3}{RE4}$ maintaining the correct low frequency common-mode gain. Differential frequency response and slew rate are unchanged by these modifications. The simulation results for the common-mode test circuit shown in Fig. 7.20 are given in Fig. 7.55. These indicate close agreement between the modular and ac Boyle macromodels.

Modifying the circuit of an existing OP AMP macromodel is at best a complex process or at worst impossible because the model details are either not known or well understood. One way to add features to an existing model is to add an external circuit to a model's terminals. This circuit acts as a signal processing element adding additional capabilities to the original macromodel. One circuit feature not modelled by any of the macromodels introduced in earlier sections is power supply rejection. By adding a simple passive electrical network to the terminals of a macromodel it is possible to model OP AMP power supply rejection. Power supply rejection(PSRR) is a measure of the ability of an OP AMP to reject unwanted signals that enter at the power terminals. It is defined as the ratio of differential-mode gain to power supply injected signal gain. The simulation of OP AMP power supply rejection[30]

[29]In the Boyle macromodel the value of CEE is set by the OP AMP slew rate. Adjusting both the positive and negative slew rates changes the value of CEE. For the UA741 these have been set at 0.625e6 and 0.500e6 respectively. This gives CEE=7.5pF which is commonly quoted for the UA741 value of CEE, see Andrei Vladimirescu, The SPICE Book,1994, John Wiley and Sons, Inc., ISBN 0-471-60926-9, pp 228-239. Also note care must be taken when choosing values for the two slew rates because negative values of CEE can occur which are physically not realisable.

[30]M. E. Brinson and D. J. Faulkner, Measurement and modelling of operational amplifier power supply rejection, Int. J. Electronics, 1995, vol. 78, NO. 4, 667-678.

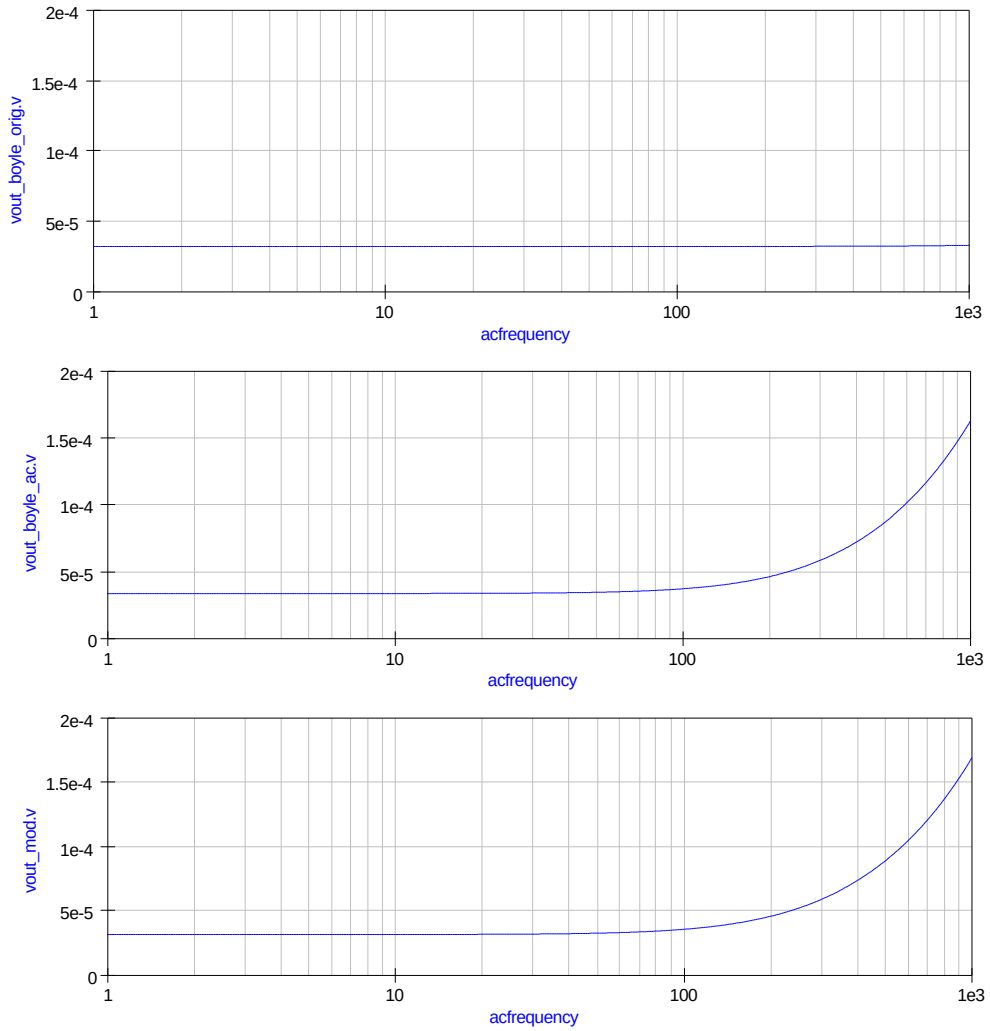


Figure 7.55: AC common-mode simulation results for (1) Boyle macromodel, (2) ac Boyle macromodel and (3) the modular macromodel

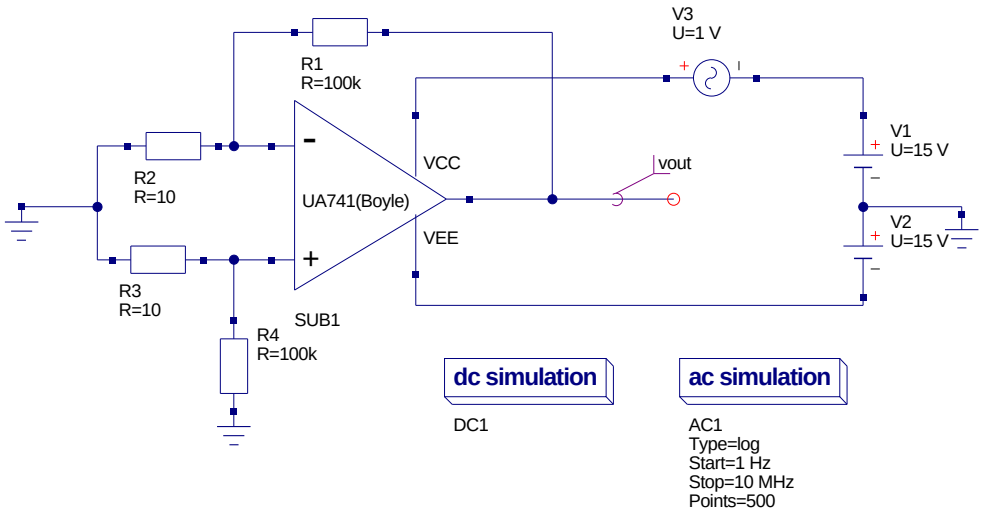


Figure 7.56: Test circuit for the simulation of PSRR(f) voltage transfer function characteristic

is possible using the test circuit given in Fig. 7.56, where

$$V_{out}(f)^{\pm} = A_D(f)\left[V^+ - V^-\right] + A_{CM}(f)\left[\frac{V^+ + V^-}{2}\right] + A_{PS}(f)^{\pm}V_S,$$

where $A_D(f)$ is the OP AMP differential-mode gain, $A_{CM}(f)$ is the OP AMP common-mode gain, and $A_{PS}(f)^{\pm}$ is the OP AMP power supply injected gain. The superscript $\pm$ indicates that the ac signal source is connected to either the OP AMP positive or negative power supply terminals but not simultaneously to both. Assuming that the OP AMP power supply injected gain has a single dominant zero at $f^{\pm}_{PSZ1}$, analysis yields

$$\frac{V_{OUT}(f)^{\pm}}{V_S} = \frac{\dfrac{1}{\alpha P_{SRR}(0)^{\pm}}\left[1 + j\left(\dfrac{f}{f^{\pm}_{PSZ1}}\right)\right]}{\left[1 + j\left(\dfrac{f}{\alpha GBP}\right)\right]}$$

Where $APS(f)^{\pm} = APS(0)^{\pm}\dfrac{\left[1 + j\left(\dfrac{f}{f^{\pm}_{PSZ1}}\right)\right]}{\left[1 + j\left(\dfrac{f}{f_{p1}}\right)\right]}$, $\alpha = \dfrac{R2}{R1 + R2} = 1e - 4$

$$PSRR(f)^{\pm} = \frac{PSRR(0)^{\pm}}{\left[1 + j\left(\dfrac{f}{f^{\pm}_{PSZ1}}\right)\right]},\ PSRR(0)^{+} = \frac{A_D(0)}{A_{PS}(0)^{+}} \text{ and } PSRR(0)^{-} = \frac{A_D(0)}{A_{PS}(0)^{-}}.$$

Typical values for the UA741 are $PSRR(0)^{+} = 110000, PSRR(0)^{-} = 170000, f^{+}_{PSZ1} = 685Hz, f^{-}_{PSZ1} = 6.2Hz$. The considerable difference in the dominant zero frequencies of the injected power supply gains is normally due to the fact that the OP AMP circuits are not symmetric when viewed from the power supply signal injection terminals. By adding

external components to an OP AMP macromodel power supply rejection effects can be easily simulated. The schematic shown in Fig. 7.57 shows the TI UA741 model with RC networks connected between the power supply terminals and earth. The voltage controlled voltage sources probe the voltages at the center nodes of the additional RC networks. These networks generate the power supply injected signals at dc. They also generate the dominant zero in the power supply rejection characteristic.

The values for the passive components can be calculated using:

$$RA = \frac{10^6}{PSRR(0)^+}, CA = \frac{1}{2 \cdot 10^6 \cdot \pi \cdot f^+_{PSZ1}}, RB = \frac{10^6}{PSRR(0)^-}, CA = \frac{1}{2 \cdot 10^6 \cdot \pi \cdot f^-_{PSZ1}}$$

Which gives, for the example UA741 device data, $RA = 9\Omega$, $CA = 232pF$, $RB = 5.9\Omega$ and $CB = 25.7pF$. Simulation waveforms for the small signal frequency response of the test circuit are shown in Fig. 7.58. In the case of the modular UA741 model the simulation signal plot clearly demonstrates the fact that the model does not correctly represent the effects due to power supply injected signals.

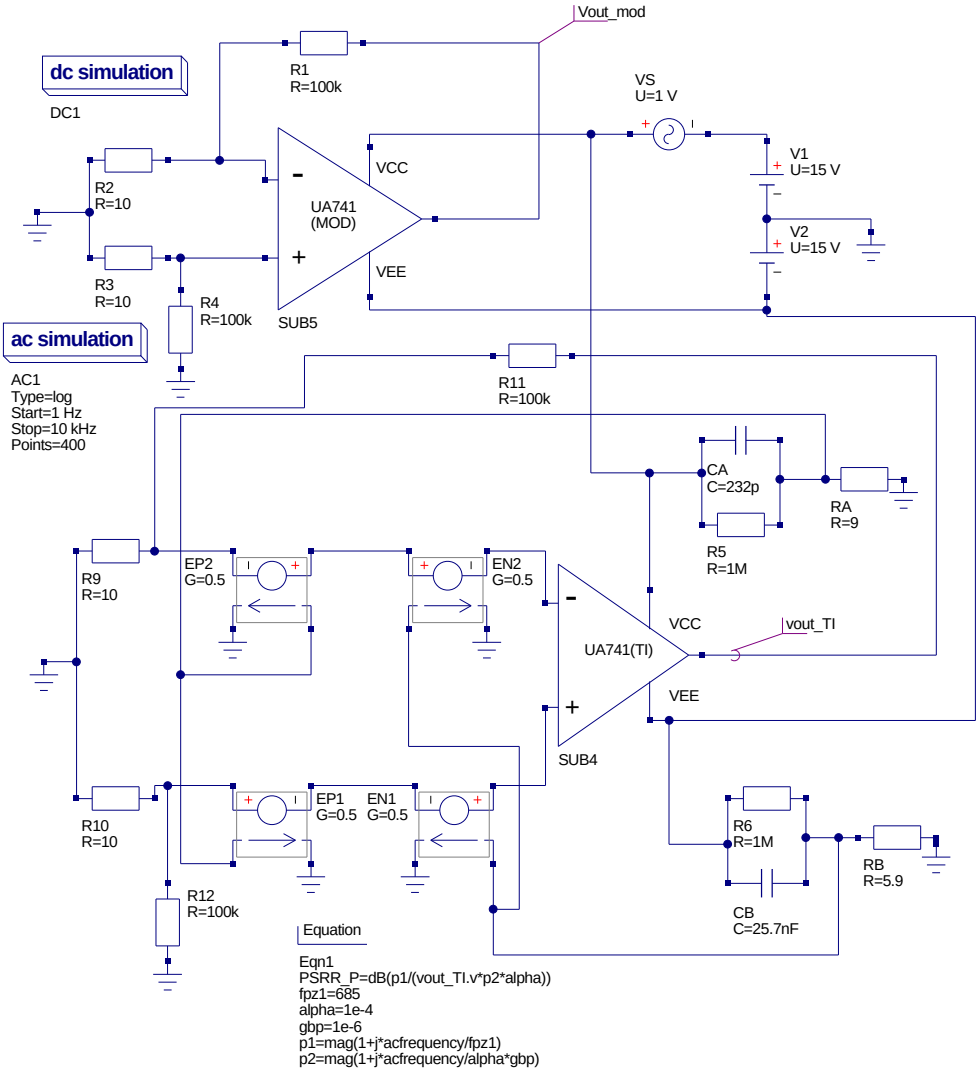


Figure 7.57: Test circuit showing OP AMP with external power supply rejection modelling network

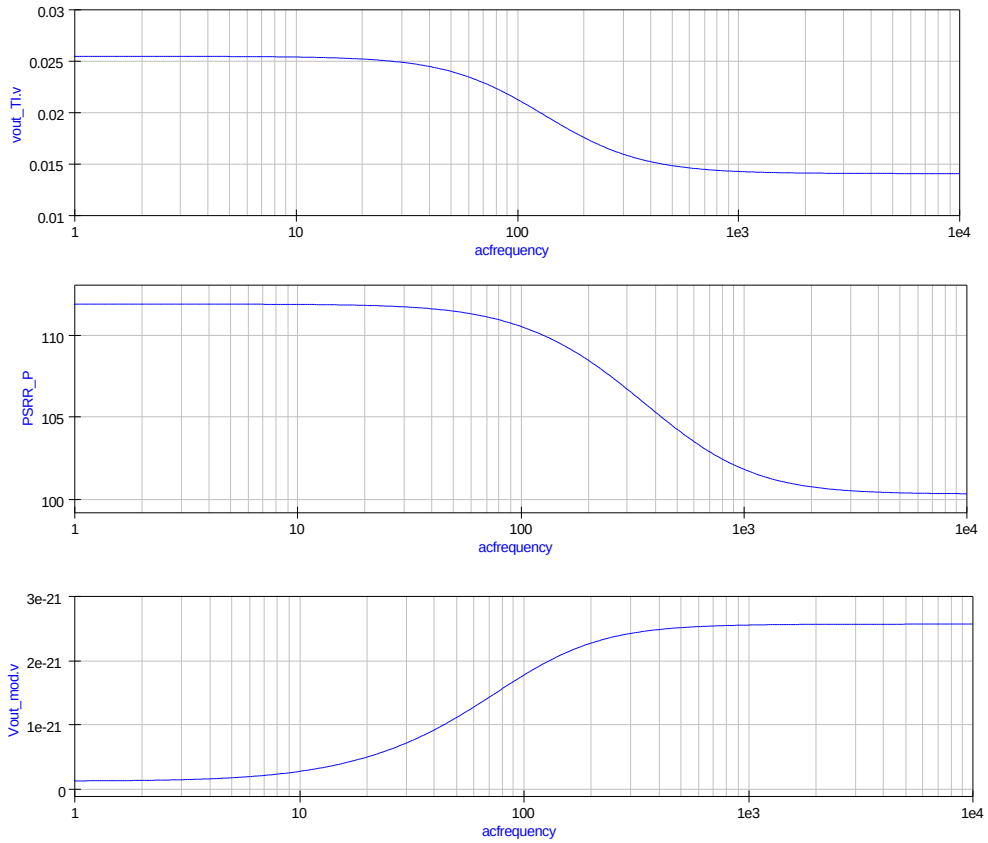


Figure 7.58: Simulation waveforms for the circuit illustrated in Fig. 7.57

7.14 End note

While writing this tutorial I have tried to demonstrate how practical models of operational amplifiers can be constructed using basic electronic concepts and the range of Qucs built-in components. The modular OP AMP macromodel was deliberately chosen as the foundation for the tutorial for two reasons; firstly Qucs is mature enough to easily simulate such models, and secondly the parameters which determine the operation of the macromodel can be be calculated directly from information provided on device data sheets. Recent modelling development by the Qucs team has concentrated on improving the SPICE to Qucs conversion facilities. This work has had a direct impact on Qucs ability to import and simulate manufacturers OP AMP models. The tutorial upgrade explains how SPICE Boyle type OP AMP macromodels can be converted to work with Qucs. The Qucs OP AMP library (OpAmps) has been extended to include models for a range of popular 8 pin DIL devices. If you require a model with a specific specification that is not modelled by an available macromodel then adding extra functionality may be the only way forward. Two procedures for extending models are outlined in the tutorial upgrade. Much work still remains to be done before Qucs can simulate a wide range of the macromodels published by device manufacturers. With the recent addition of subcircuit/component equations to Qucs it is now possible to write generalised macromodel macros for OP AMPs. However, before this can be done time is required to fully test the features that Stefan and Michael have recently added to Qucs release 0.0.11. This topic and the modelling of other OP AMP properties such as noise will be the subject of a further OP AMP tutorial update sometime in the future. My thanks to David Faulkner for all his help and support during the period we were working on a number of the concepts that form part of the basis of this tutorial. Once again a special thanks to Michael Margraf and Stefan Jahn for all their help and encouragement over the period that I have been writing this tutorial and testing the many examples it includes.

8 Modelling the 555 Timer

8.1 Introduction

The 555 timer was designed by Hans R. Camenzind in 1970[1] and first produced by Signetics during the period 1971-1972[2]. The device was originally called "The IC time machine" and given the part number SE555/NE555. Over the last 30 plus years more than ten different semiconductor chip production companies have made 555 parts, making it one of the most popular ICs of all time[3]. Today it is still used in a wide range of circuit applications.

The 555 timer is one of the first examples of a mixed mode IC circuit that includes both analogue and digital components. The primary purpose of the 555 timer is the generation of accurately timed single pulse or oscillatory pulse waveforms. By adding one or two external resistors and one capacitor the device can function as a monostable or astable pulse oscillator.

The 555 timer is a difficult device to simulate. During circuit operation it switches rapidly between two very different DC states[4]. Such rapid changes can be the cause of simulator DC convergence and transient analysis errors. Most of the popular simulators include some form of 555 timer model, either built-in or as a subcircuit, which functions to some degree. These models usually include a number of p-n junctions and non-linear controlled sources, making simulation times longer than those obtained with simpler models. At the heart of the 555 timer are two comparators and a set-reset flip flop. A block diagram of the main functional elements that comprise the 555 timer is illustrated in Fig. 8.1.

The current Qucs release does not include a model for the 555 timer. The purpose of the work reported in this tutorial note has been to develop a 555 timer model from scratch which simulates efficiently, and is based only on the circuit components implemented in Qucs 0.0.10. Moreover, while developing the Qucs 555 model every attempt has been made to reduce the number of p-n junctions to a minimum, yielding both model simplicity

[1] See "The 555 Timer IC. An interview with Hans Camenzind - The designer of the most successful integrated circuit ever developed", `http://semiconductormuseum.com/Transistors/LectureHall/Camenzind/`

[2] Now part of the Philips organisation.

[3] Recent manufacturing volumes indicate that the 555 timer is as popular as ever, with for example, Samsung (Korea) producing over one billion devices in 2003; see Wikipedia entry at `http://en.wikipedia.org/`

[4] Typically between ground and a voltage close to power rail VCC.

and reduced circuit simulation times. The approach adopted is centred on established macromodelling techniques where signals at the timer device pins accurately model real device signals but internal macromodel signals often bare no relation to those found in an actual device. Internally, the macromodel simply processes input signal information and outputs signals, in the correct format, to the device output pins. In no way is an attempt made to simulate the actual 555 timer circuitry.

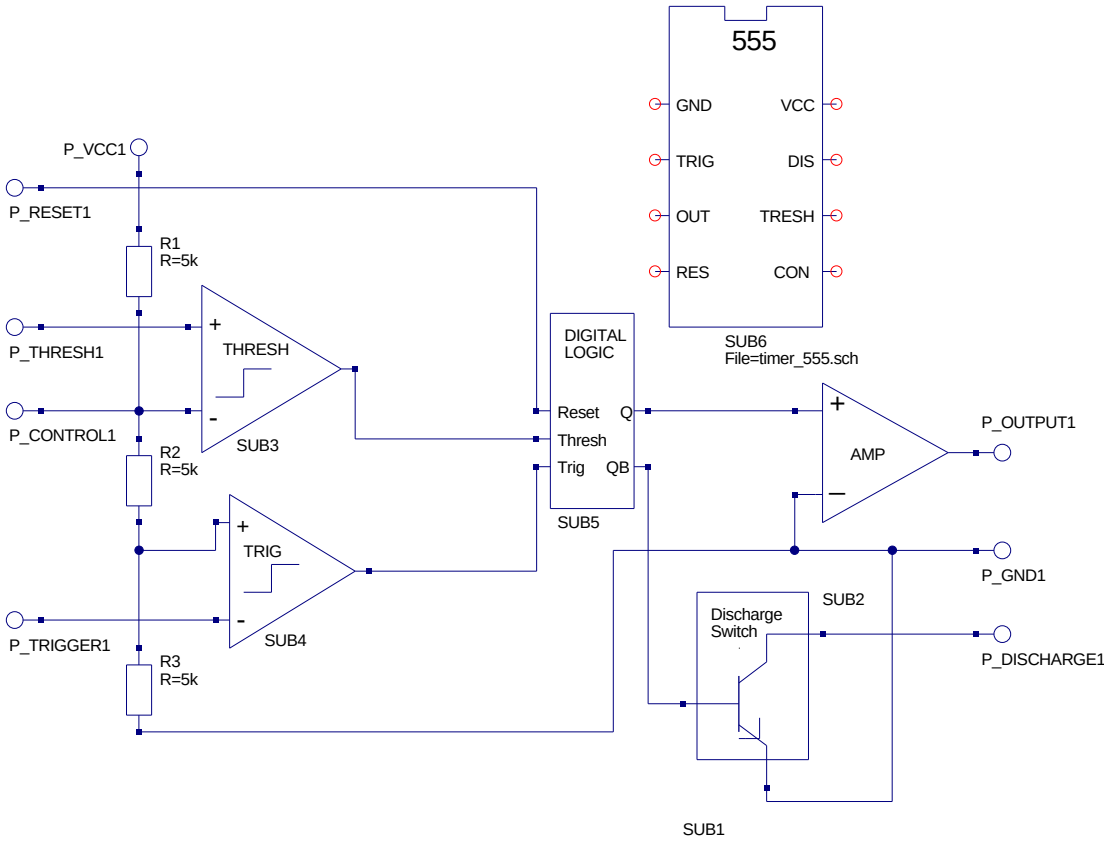


Figure 8.1: 555 Timer functional block diagram.

8.2 The Qucs 555 timer model

Fig. 8.1 illustrates the new Qucs 555 timer model. In this model each of the major functional blocks have been separated into macromodel subcircuits, grouping similar types of component together. Essentially, the model only includes standard Qucs components which all work together to produce the correct output signals through careful selection of threshold parameters, voltage limits, logic levels and rise and fall times. These notes concentrate on explaining the structure and parameters of the macromodel subcircuits that form the

555 timer model, rather than describing the function of the device[5]. The 555 timer is an 8 pin device with:

- Pin 1 Ground [GND] - Most negative supply connected to the device, normally this is common ground (0V).
- Pin 2 Trigger [TRIG] - Input pin to the lower comparator. Used to set the RS latch.
- Pin 3 Output [OUT] - The 555 timer output signal pin.
- Pin 4 Reset [RES] - Used to reset the RS latch.
- Pin 5 Control [CON] - Direct access point to the (2/3)VCC divider node. Used to set the reference voltage for the upper comparator.
- Pin 6 Threshold [THRESH] - Input pin to upper comparator. Used to reset the RS latch.
- Pin 7 Discharge [DIS] - Collector output of an npn BJT switch. Used to discharge the external timing capacitor.
- Pin 8 VCC [VCC] - Most positive supply connected to device, normally this is 5V, 10V or 15V.

8.2.1 The trigger comparator macromodel

The trigger comparator input pins are connected between the (1/3)VCC divider node and device package pin 2 (TRIG). Trigger input signals dropping below the (1/3)VCC divider node voltage cause the trigger output voltage to switch, setting the RS latch in the digital logic subcircuit. This action also causes the 555 timer output signal to go high. The trigger input is level sensitive. Retriggering will occur if the trigger pulse is held low longer than the 555 timer output pulse width. The trigger comparator circuitry also has a storage time of several microseconds, limiting the minimum monostable output pulse to around 10μS. A DC current, popularly referred to as the trigger current, flows from device pin 2 (TRIG) into the external circuit. This has a typical value of 500 nA, setting the upper limit of resistance that can be connected from pin 2 to ground[6]. The circuit diagram of the trigger comparator macromodel is shown in Fig. 8.2. The differential input signal is sensed by operational amplifier OP1. This has it's gain set to 1e6, giving a differential input signal resolution of 1μV. OP1 output voltages are limited to $\pm$1V. Note the upper +1V signal level corresponds to a logic '1' signal. Finally, the trigger comparator output voltage rise and fall times are set by time constant $R1 * C1$. This network also adds a time delay to the comparator macromodel.

[5]A good tutorial guide to the operation of the 555 timer can be found at `http://www.uoguelph.ca/~antoon/gadgets/555/555.html`

[6]At VCC = 5V this resistance is roughly 3.3MΩ.

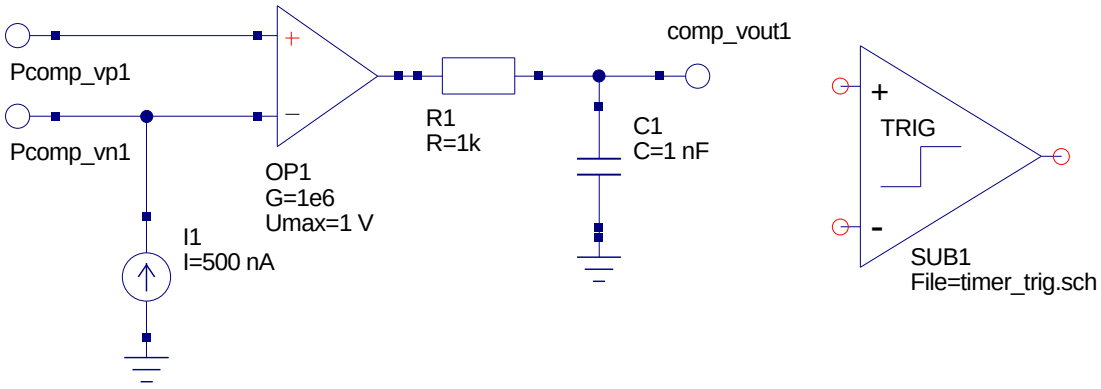


Figure 8.2: Trigger comparator macromodel.

8.2.2 The threshold comparator macromodel

The threshold comparator macromodel is shown in Fig. 8.3. It is very similar to the trigger comparator macromodel; one noteable difference is the size and direction of pin 6 (THRES) threshold DC current which is typically 100nA and flows into pin 6 from the external circuitry[7]. The threshold comparator is used to reset the RS latch in the 555 timer digital logic block, causing the 555 timer output to go low. Resetting occurs when the signal applied to external pin 6 (THRES) is driven from below to above the (2/3)VCC divider node voltage. Again the threshold input is level sensitive.

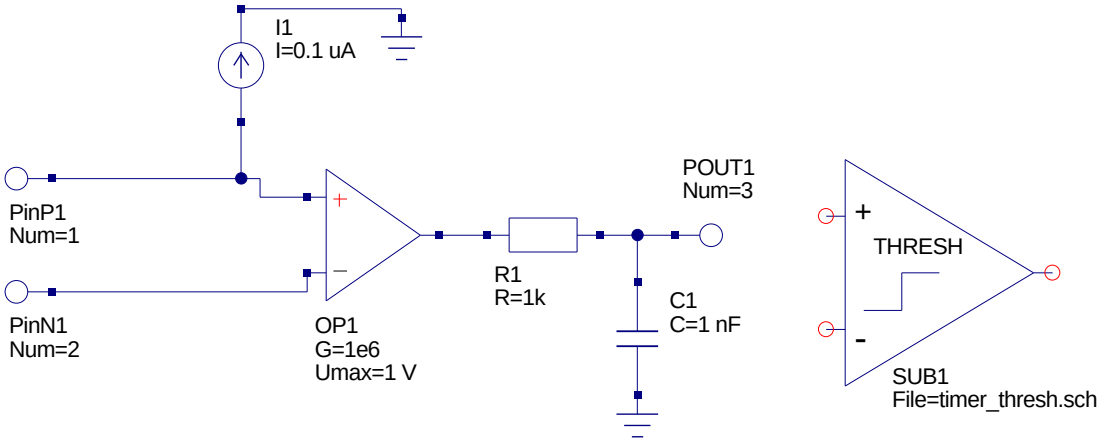


Figure 8.3: Threshold comparator macromodel.

[7]The threshold DC current sets the upper limit to the value of the external resistor that can be connected between pin 6 and the VCC supply - for VCC = 5V this is approximately 16MΩ, with VCC = 15 V this rises to roughly 20MΩ.

Set (S)	Reset (R)	Q (P-Q1)	QB (P-QB1)	Notes
1	0	1	0	Set state
0	0	1	0	
0	1	0	1	Reset state
0	0	0	1	
1	1	0	0	Undefined

Table 8.1: Truth table for an SR latch constructed using NOR gates.

8.2.3 The digital logic macromodel

The digital logic macromodel consists of an SR latch with additional combinational gates at the input of the model, see Fig. 8.4. The truth table for the SR latch is listed in Table 8.1. All gates in the macromodel have logic '1' set at 1V and logic '0' set at 0V. RC timing networks have been added to the output of each gate, ensuring that the gates have a finite rise and fall times rather than the Qucs default value of zero seconds[8]. Gate input signals with values less than the gate threshold voltage (0.5V) are considered to be a logic '0' signal. A logic '0' signal on 555 timer pin 4 (RES) also resets the SR latch causing the output signal, pin 3 (OUT), to move to a low state. The reset signal is an override signal in that it forces the timer output to a low state regardless of the signals on other timer input pins. Reset has a delay time of roughly 0.5μS, making the minimum reset pulse width of approximately 0.5μS. The reset signal is inverted then ORed with the threshold comparator output signal.

[8]In mixed mode circuit simulation transient analysis problems can occur when devices change state in zero seconds, see later notes for comments on this topic.

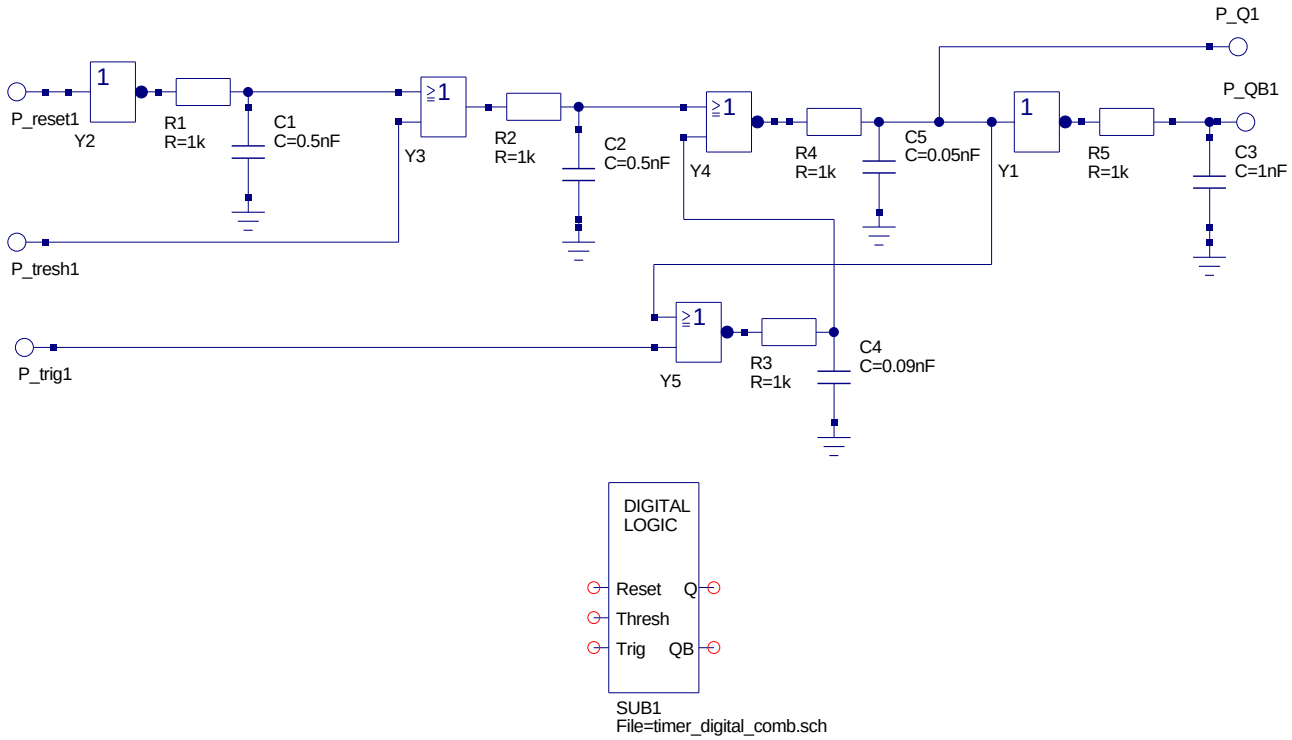


Figure 8.4: Digital logic macromodel.

8.2.4 The 555 timer output amplifier macromodel

Illustrated in Fig. 8.5 is the macromodel for the timer output amplifier. This is a simple model constructed from a voltage gain block plus a resistor to represent the 555 timer output resistance. The voltage gain block has it's value set to 3.5 in Fig. 8.5. This is the value needed to scale the logic '1' signal voltage to the required external voltage at timer output pin 3 (OUT). This value is only correct for power supply voltage VCC set to 5V, and must be changed for other voltages[9].

[9]At this time Qucs does not allow parameters to be passed to subcircuits, making it difficult to write generalised macromodels. Adding parameter passing to subcircuits and the calculation of component values using equations is on the to-do list. Suggested values for the amplifier gain are: (1) VCC = 5V, G = 3.5, (2) VCC = 10V, G = 8.5V and (3) VCC = 15V, G = 13.5. These gain values correct for the voltage drop in the 555 timer totem-pole output stage.

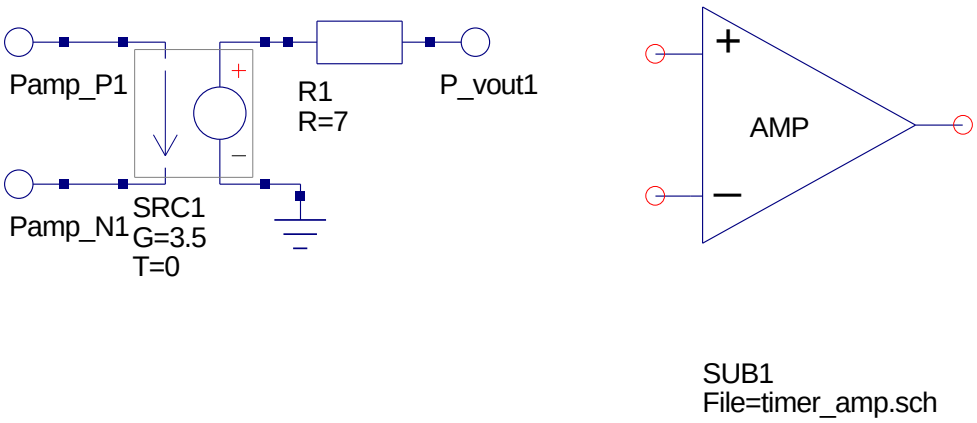


Figure 8.5: Output amplifier macromodel.

8.2.5 The discharge switch macromodel

The discharge switch macromodel is shown in Fig. 8.6. Like the actual 555 timer the macromodel discharge switch is based on an npn transistor. A logic '1' signal applied to terminal `pin_control_in1` turns the npn transistor on causing the path from the collector (555 timer pin DIS) to ground to become low resistance. It is through this branch that the timer external capacitor is discharged. The reverse characteristic is observed when the input control voltage is logic '0'. In this case the collector to ground branch has a very high resistance. Resistor R1 is included in the macromodel to limit the npn base current when the BJT is turned on. Similarly, resistor R2 has been added to the model to limit the external capacitor discharge current[10].

[10]Normally the external timing capacitor is discharged through a resistor in series with the collector to ground path. However, if this series resistor is very small, or indeed does not exist, it is theoretically possible for the discharge current to become very large, which in turn leads to DC convergence errors or very long transient simulation times.

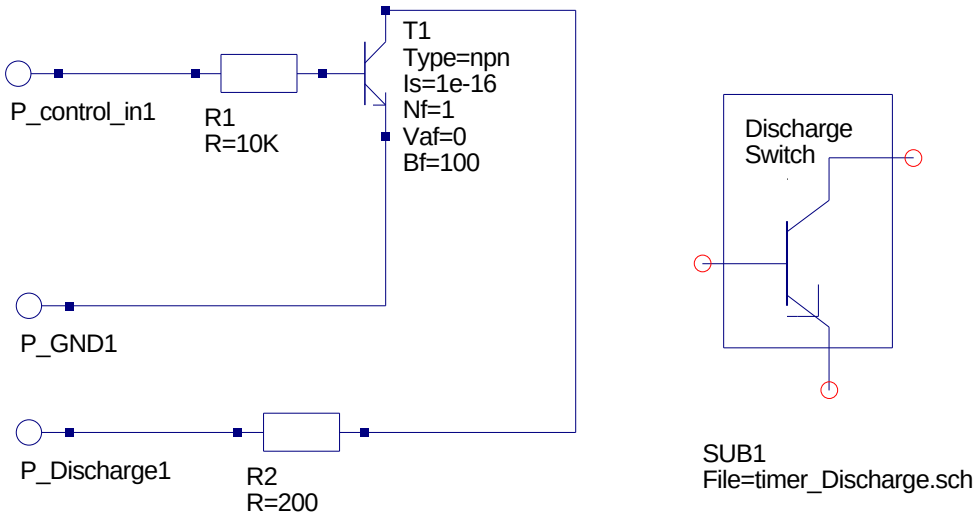


Figure 8.6: The discharge switch macromodel.

8.3 Published 555 timer test circuits

The majority of manufacturers outline in their 555 timer specification sheets a range of fundamental circuit applications[11]. A number of these circuits are introduced as a series of simulation test cases. The conditions chosen for the simulation tests are as follows:

- Integration method Gear, order 6 (this method works well with circuits that contain time constants that have widely different values)[12].
- Input driver signals have a finite rise and fall time, usually in nano seconds (problems can occur when driver signals have either zero or very small rise and fall times - often a simulator will reduce the transient analysis step size in an attempt to reduce errors which in turn can significantly increase simulation run times).
- Transient simulation parameter MinStep is set to one hundredth, or less, of the smallest rise or fall time in the circuit (this is a good rule of thumb, giving reasonable simulation times and accuracy, normally without DC convergence or transient analysis time step problems).

8.3.1 The 555 timer monostable pulse generator

Figure 8.7 shows the basic 555 timer monostable pulse generator circuit. The output pulse width is given by the equation $T = 1.1 * R5 * C1$; when $R5 = 9.1k$ and $C1 = 0.01\mu F, T = 1ms$. Figure 8.8 illustrates the simulation waveforms for the monostable oscillator.

[11] See for example the "Applications Information" section of the National Semiconductor LM555 Timer data sheet, July 2006, www.national.com.

[12] One of the simulation tests also presents results using the standard trapezoidal second order integration method.

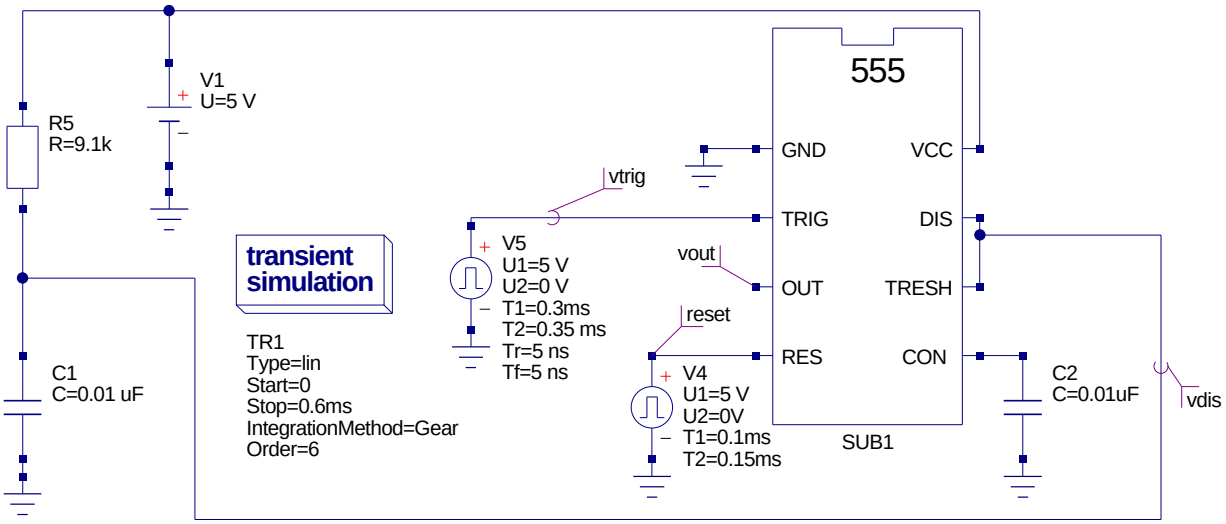


Figure 8.7: The basic 555 timer monostable pulse generator.

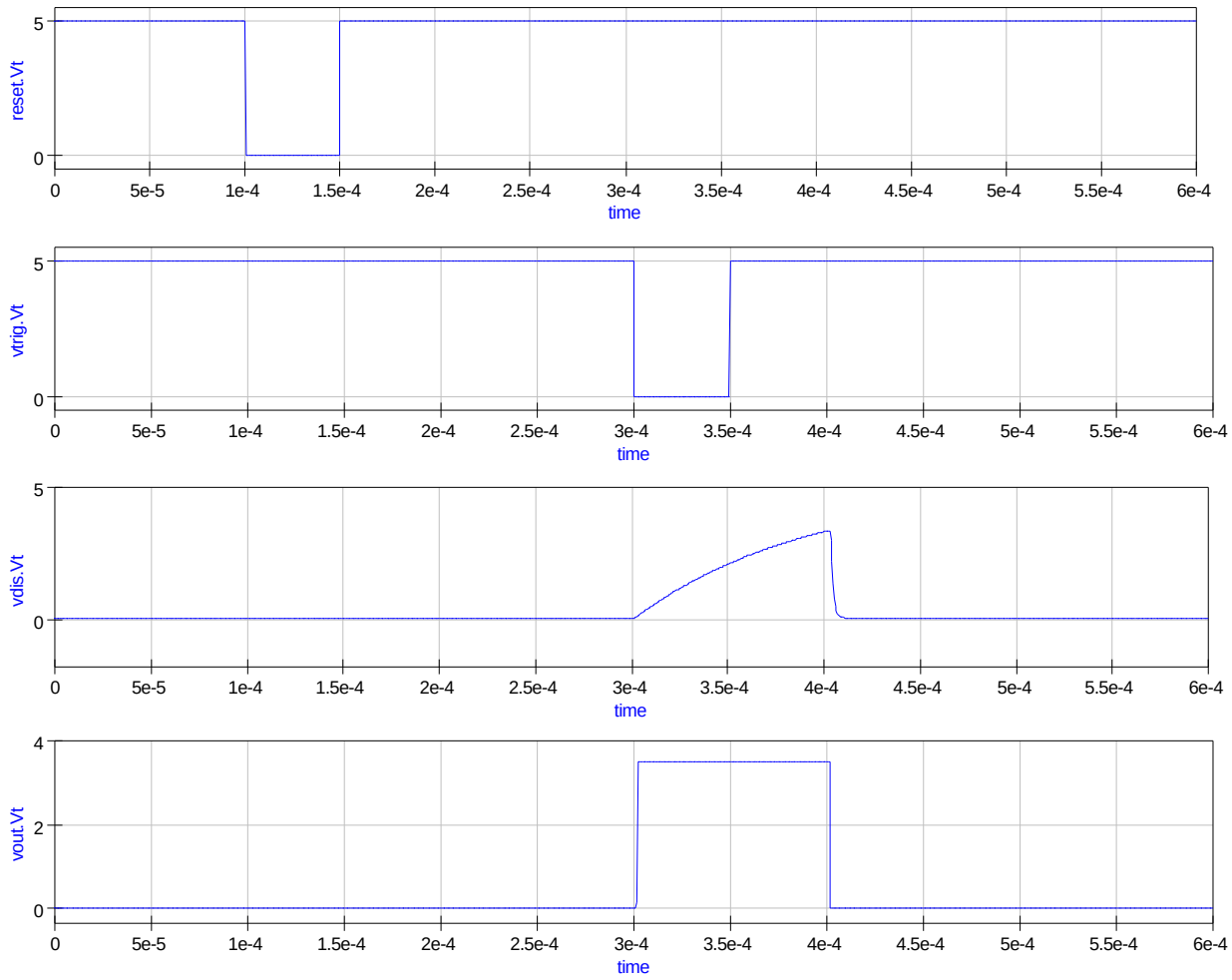


Figure 8.8: Simulation waveforms for the basic monostable pulse generator.

8.3.2 The 555 timer astable pulse oscillator

Figure 8.9 shows the basic 555 timer astable pulse generator circuit. The charging time for capacitor C1 is given by $tc = 0.693(R5 + R6)C1$ seconds, and the discharge time by $td = 0.693(R6)C1$ seconds. Hence, the period and frequency of oscillation are:
$T = tc + td = 0.693(R5 + 2R6)C1$ seconds, and $f = \dfrac{1.44}{(R5 + 2R6)C1}$ Hz.

The duty cycle for the timer output waveform is also given by $D = \dfrac{R6}{R5 + 2R6}$.

Figure 8.10 illustrates the simulation waveforms for the astable oscillator. When resistor R6 is shunted by a diode, capacitor C1 charges via resistor R5 and discharges via resistor R6. On setting R5 = R6 a 50 percent duty cycle results[13], see Figures 8.11 and 8.12.

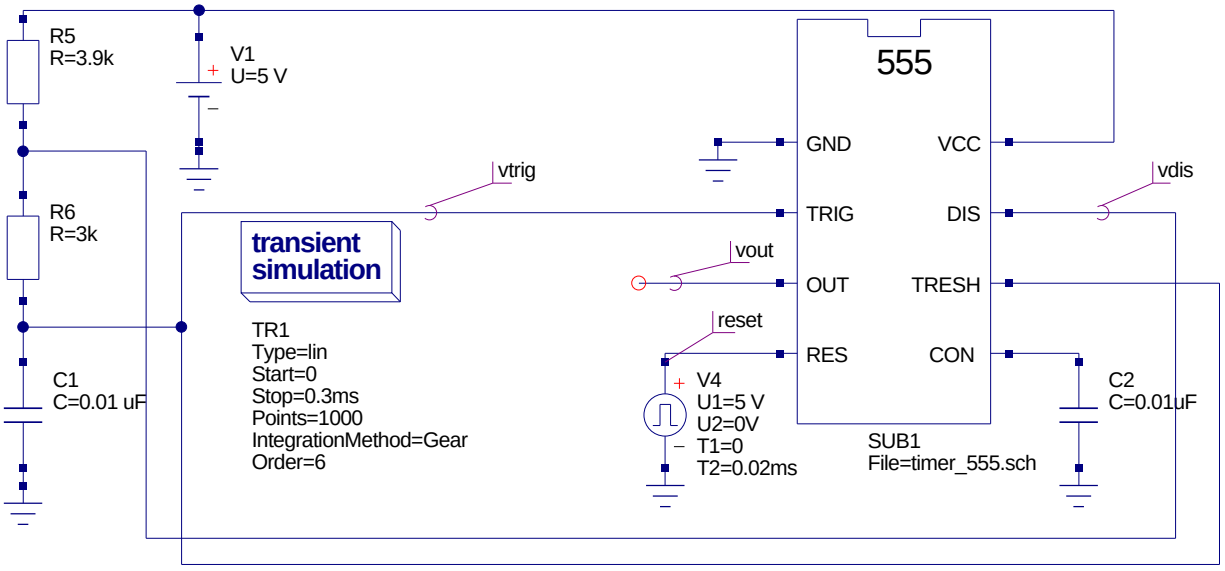


Figure 8.9: The basic 555 timer astable pulse generator.

[13]The value of R6 needs to be trimmed to set the duty cycle to exactly 50 percent.

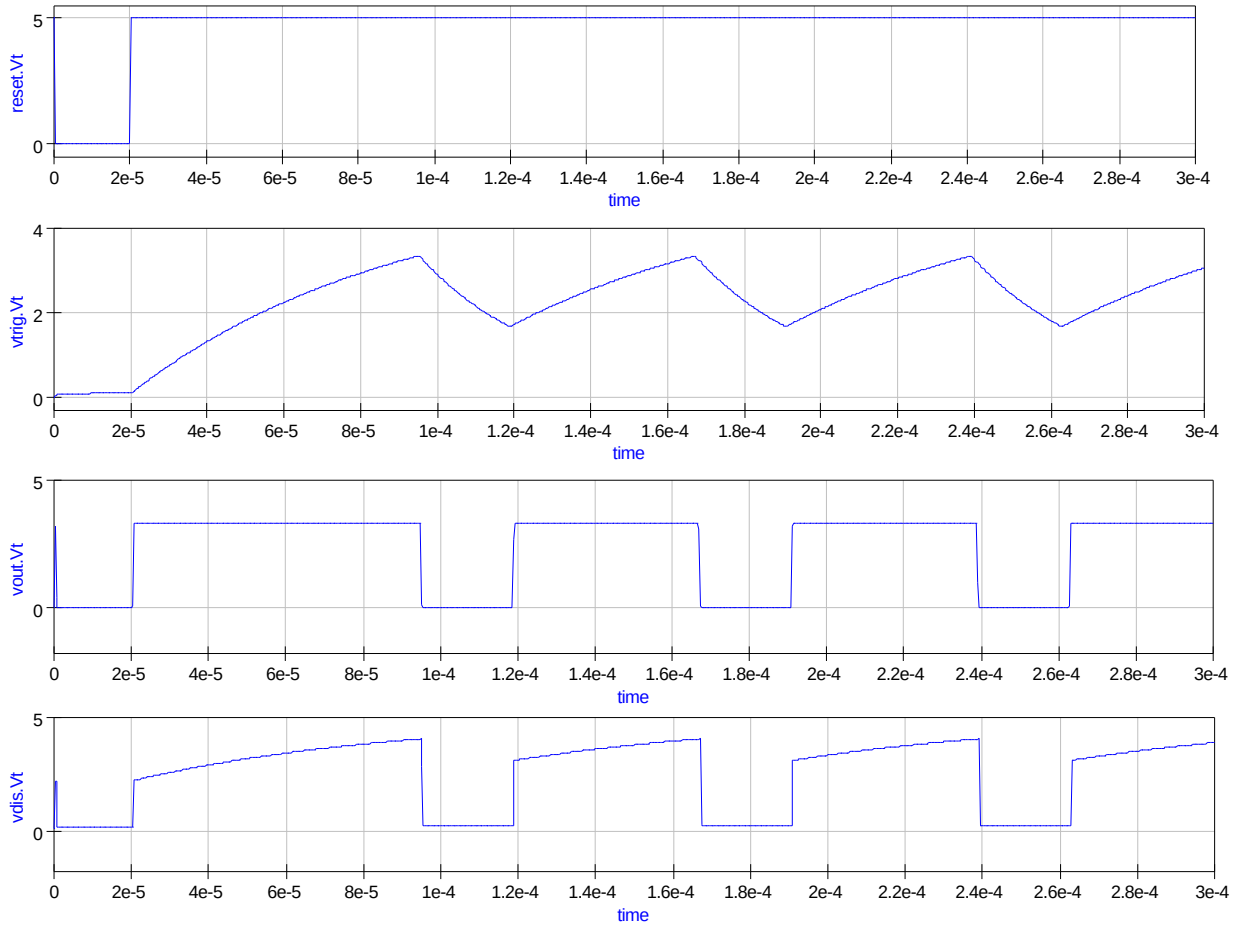


Figure 8.10: Simulation waveforms for the basic astable pulse generator.

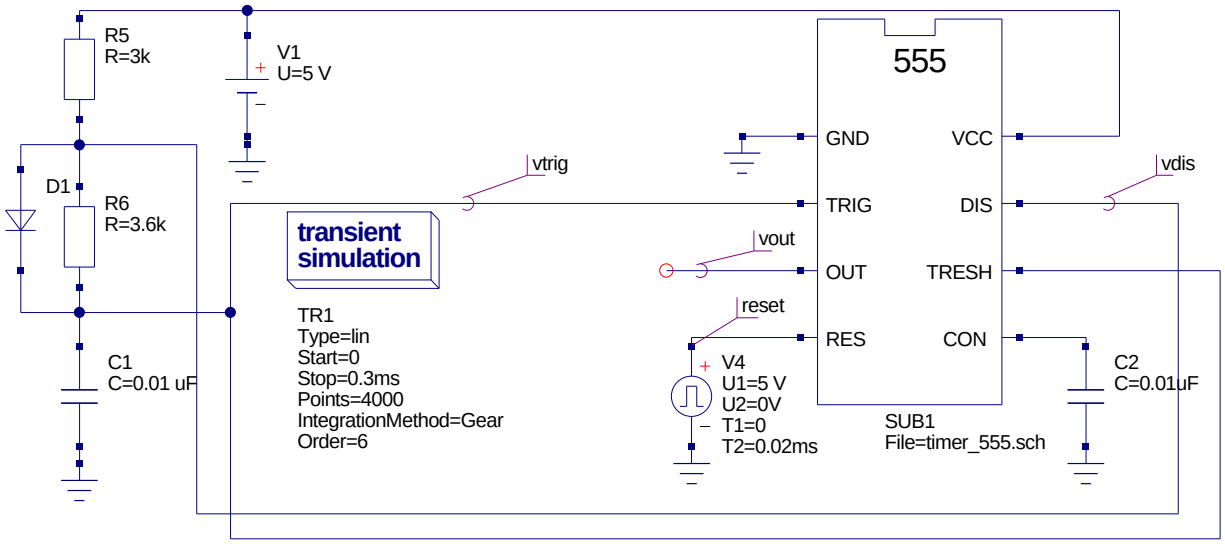


Figure 8.11: 555 timer astable pulse generator with 50 percent duty cycle.

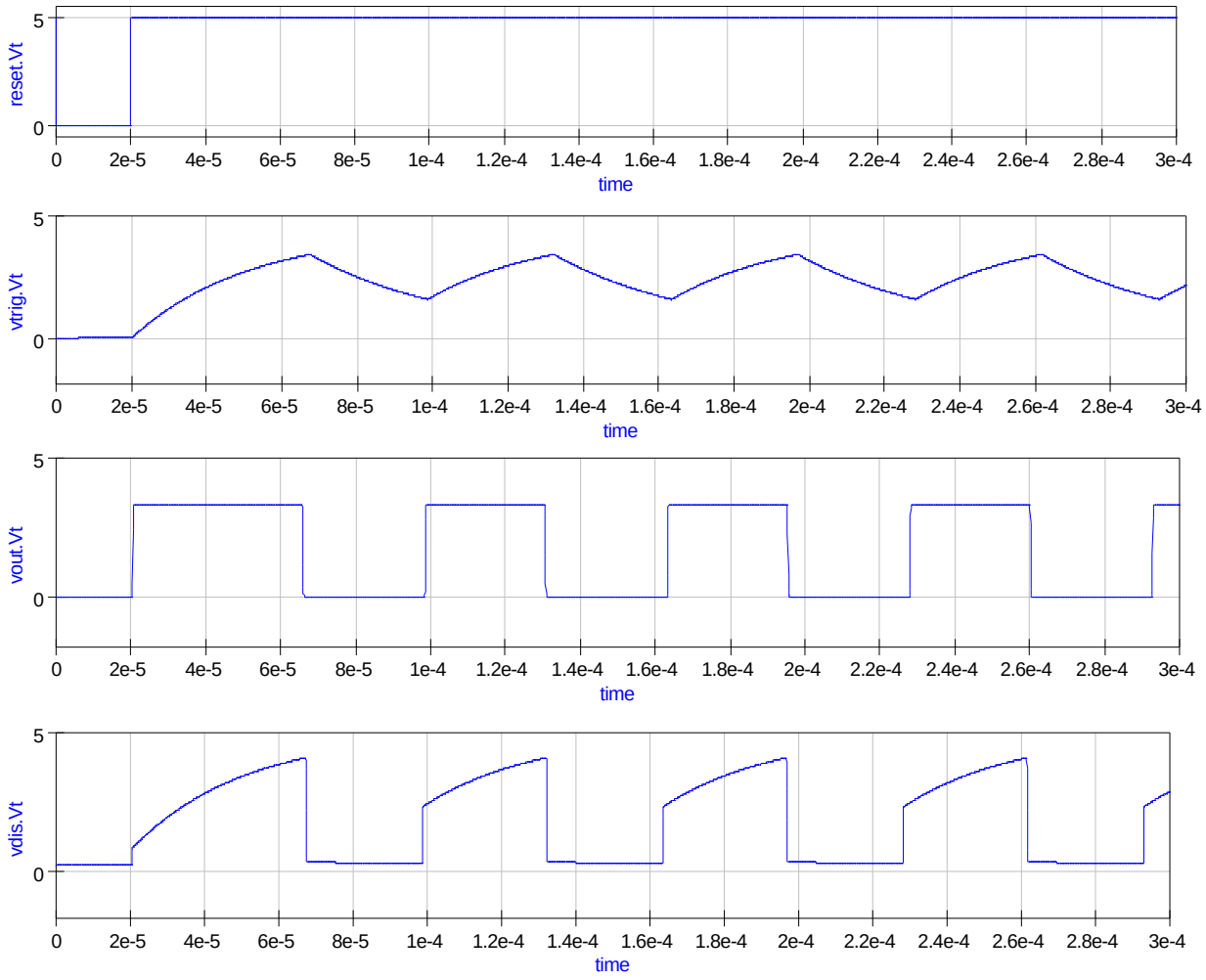


Figure 8.12: Simulation waveforms for 50 percent duty cycle astable pulse generator.

8.3.3 Pulse width modulation

Triggering the 555 timer in monostable mode with a continuous sequence of pulses allows the output pulse width to be modulated by changing the amplitude of a signal applied to the control input pin 5 (CON). An example pulse width modulator circuit is given in Fig. 8.13. In this circuit components C2, R6 and D1 convert the 555 trigger signal into a falling edge triggering signal. This can be seen in Fig. 8.14 which illustrates the trigger, discharge and resulting output waveform. The 555 timer control pin is driven from a voltage pulse source. The specification of the control waveform has been chosen to generate a triangular shaped signal so that the modulation of the pulse width can be clearly seen as the control signal amplitude changes.

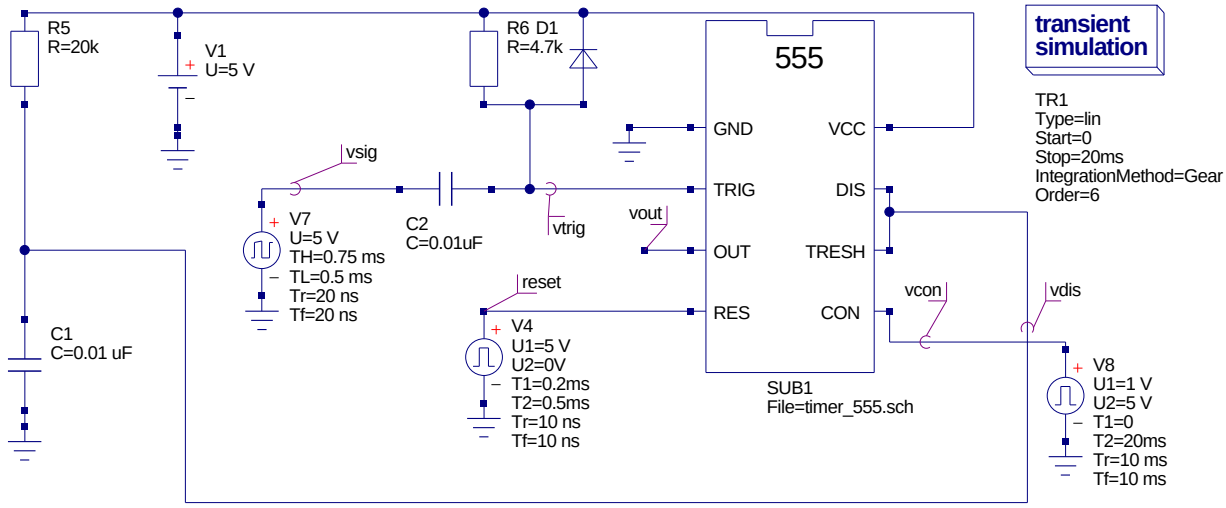


Figure 8.13: Pulse width modulator 555 timer circuit.

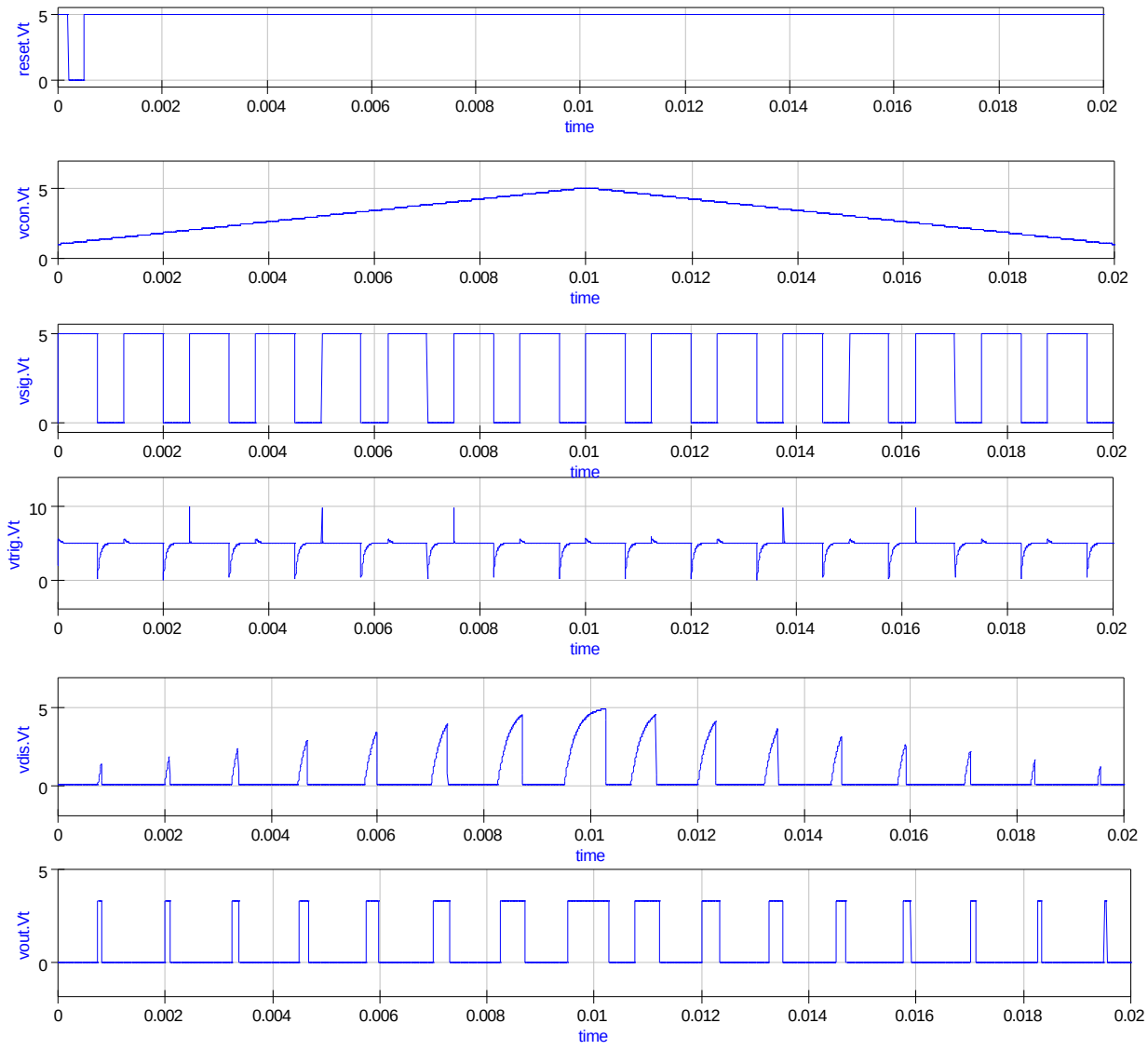


Figure 8.14: Simulation waveforms for pulse width modulator.

8.3.4 Pulse position modulation

A pulse position modulator can be constructed from the astable waveform generator given in Fig. 8.9. A modulating signal is applied to the control input pin 5 (CON); see Fig. 8.15. This signal causes the pulse position to vary with the amplitude of the applied modulating signal. A typical set of simulation waveforms for this circuit are shown in Fig. 8.16. This is a very difficult circuit to simulate. It is one case where the trapezoidal integration method works successfully whereas the 6th order Gear integration method appears to fail[14]. Note that the trapezoidal results were obtained using 30000 points, Initial step = 0.001 nS, MinStep = 1e-16, MaxIter = 5000, abstol = 10uA and vntol = 10uV.

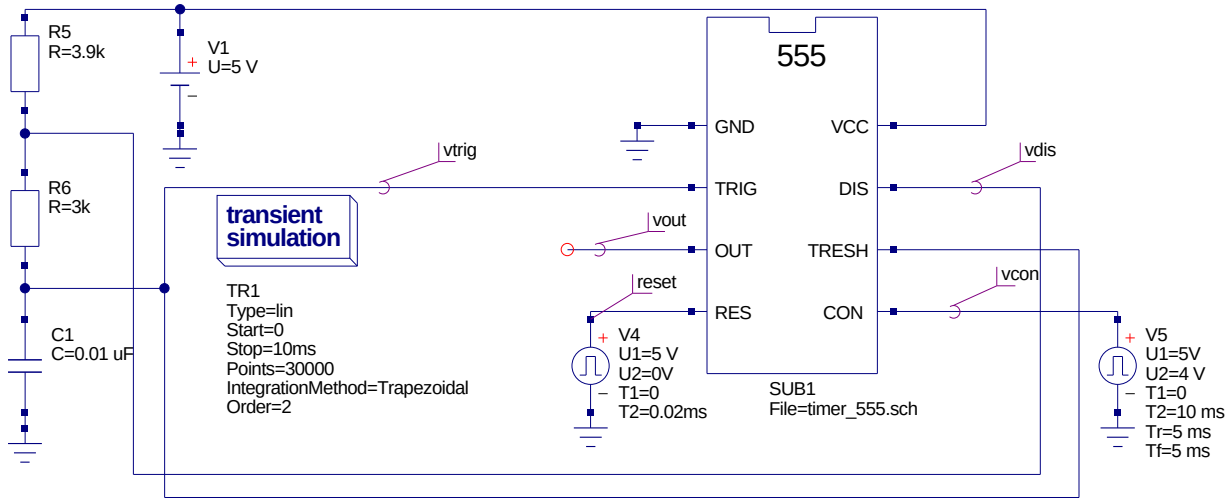


Figure 8.15: Pulse position modulator 555 timer circuit.

[14]The transient simulation never finishes and can only be terminated by clicking the simulation abort button.

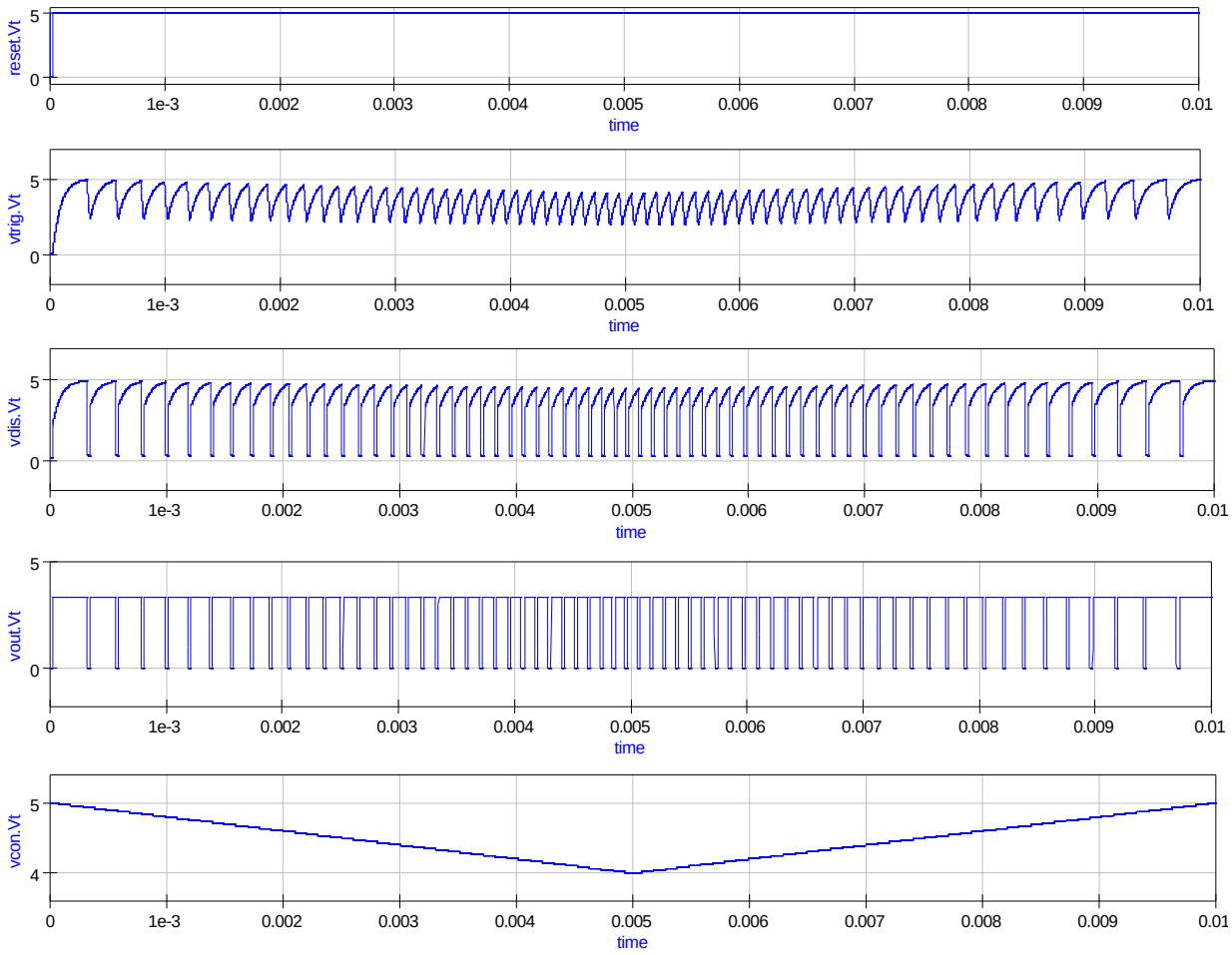


Figure 8.16: Simulation waveforms for pulse position modulator obtained using trapezoidal integration.

8.4 Multiple 555 timer simulation examples

Having established in the last section that the new Qucs 555 timer model can simulate the standard application circuits listed in a typical device data sheet, this part of the tutorial introduces two further, more complex, examples that demonstrate how the 555 timer is used in practice.

8.4.1 Sequential pulse train generation

A very practical application of the 555 timer is the generation of timing pulses for control purposes. The circuit illustrated in Fig. 8.17 shows a set of monostable pulse generators connected in series and parallel. After circuit reset the falling edge of input pulse vin triggers the start of pulse sequence generation. The time duration of each monostable

pulse is set by external capacitors C1 to C4[15]. The specification of the monostable pulse generator subcircuit is given in Fig. 8.18. The sequential pulse generator is a complex circuit with:

```
60 R instances, 40 C instances, 4 VCVS instances, 1 Vdc instances,
8 Idc instances, 2 Vpulse instances, 8 OpAmp instances, 4 Diode instances,
4 BJT instances, 8 Inv instances, 8 NOR instances and 4 OR instances.
```

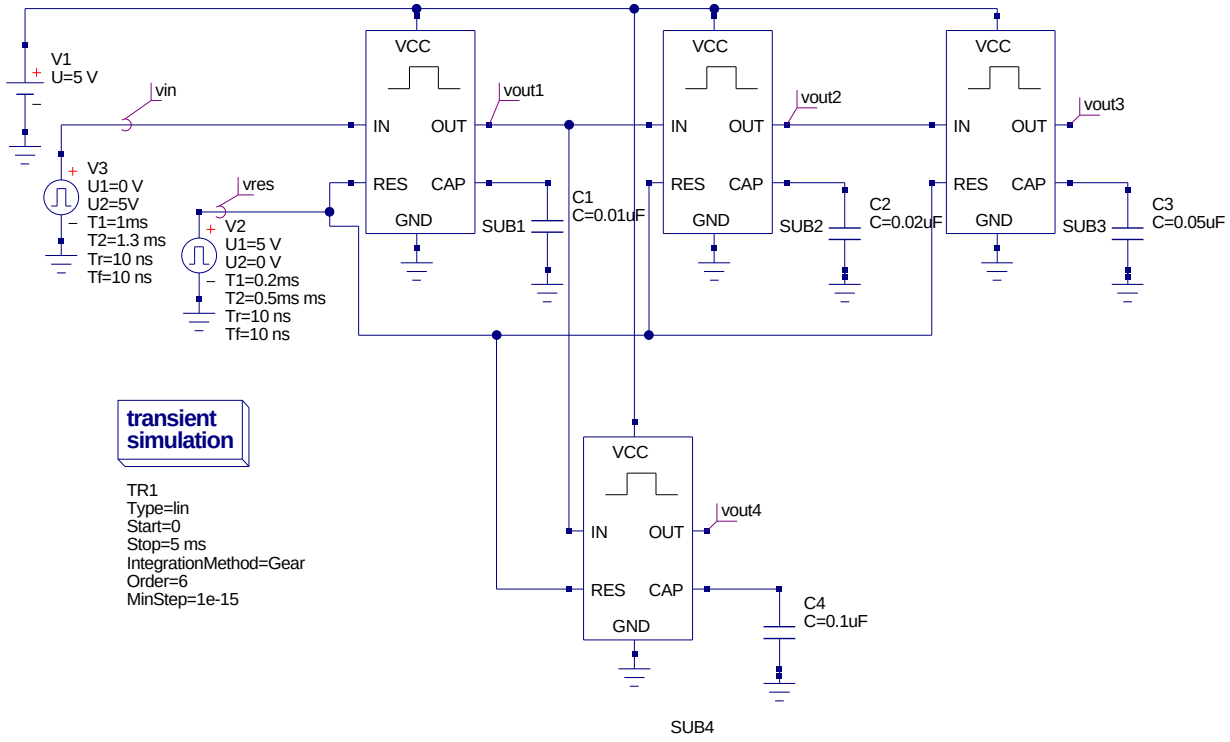


Figure 8.17: Sequential pulse generator circuit.

[15]The pulse duration times set by C1 to C4, in Fig. 8.17, have simply been chosen for demonstration purposes and do not represent any particular control timing sequence.

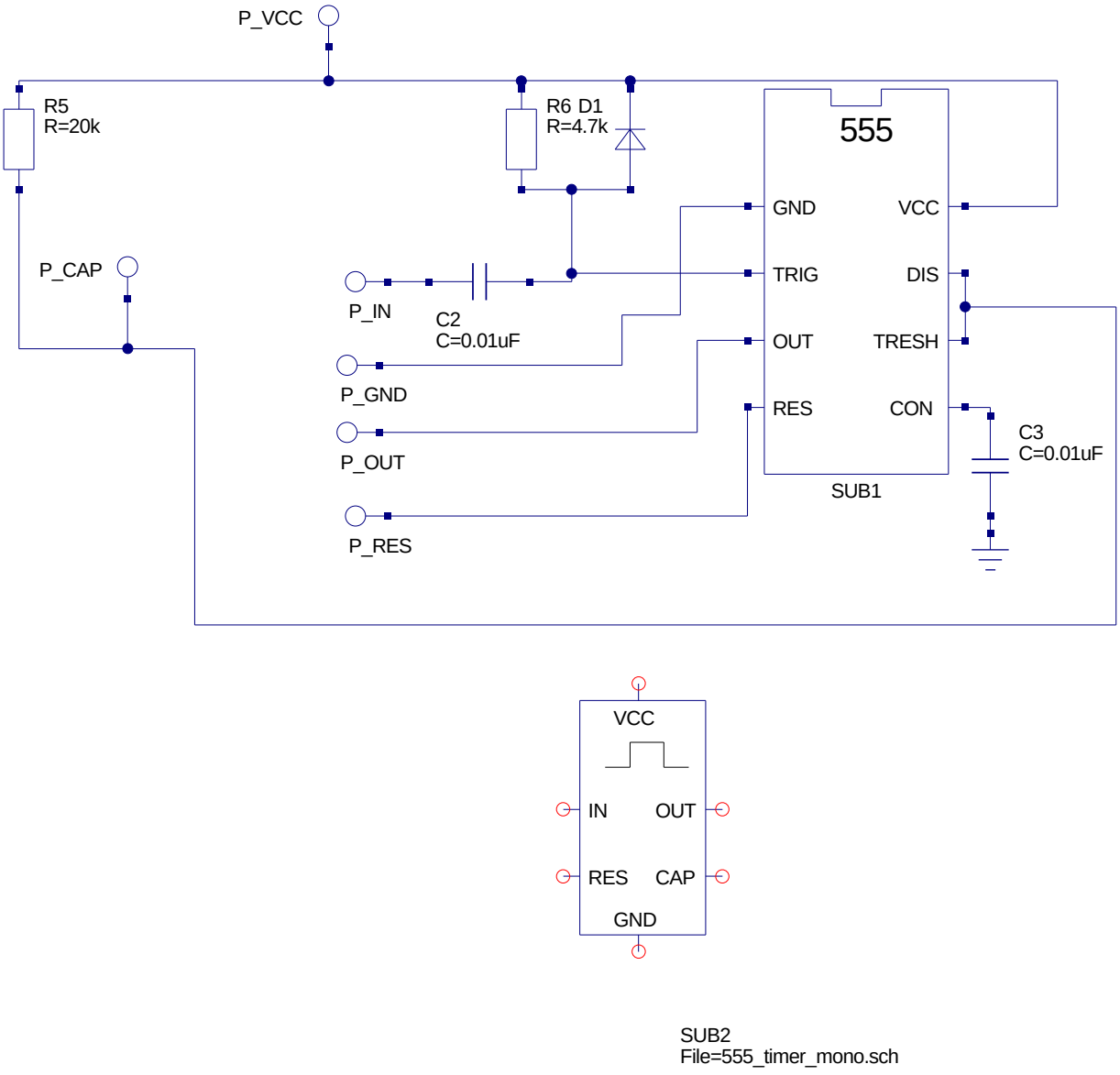


Figure 8.18: Monostable pulse generator subcircuit.

The large number of components, and indeed the complexity of the circuit, tend to make the simulation time of the pulse train generator circuit much greater than typical times recorded when simulating single 555 timer circuits. Also, circuit DC convergence and transient analysis time step errors can be a problem, due to switching discontinuities, making careful selection of the non-linear diode parameters and the transient analysis conditions essential. In Fig. 8.18 a diode is used to clamp the 555 timer trigger input at five volts when the signal attempts to rise above 5 volts. The default Qucs diode parameters are similar to those specified by SPICE[16]. By default the diode emission constant is set to 1 and the diode series resistance to zero ohms. Neither of these values are particularly representative for silicon diodes. For silicon devices, rather than germanium diodes, n needs to be between roughly 1.5 and 2. Similarly, all diodes have some series resistance, often in the range 0.1 to 10 ohms depending on the power rating of the diode. To aid simulation these parameters have been set to n = 2 and Rs = 10Ω. Figure. 8.19 illustrates a typical set of signal waveforms obtained from the simulation of the sequential pulse generator: the simulation conditions employed to generate these results are; Integration method = Gear, Order = 6, initialStep = 1 ns, MinStep = 1e-15, reltol = 0.001, abstol = 10μA, vntol = 10μV, Solver = CroutLU and initialDC = yes.

[16]The default values were set in an early version of SPICE, probably version 1, and appear to have not been changed as the simulator was developed.

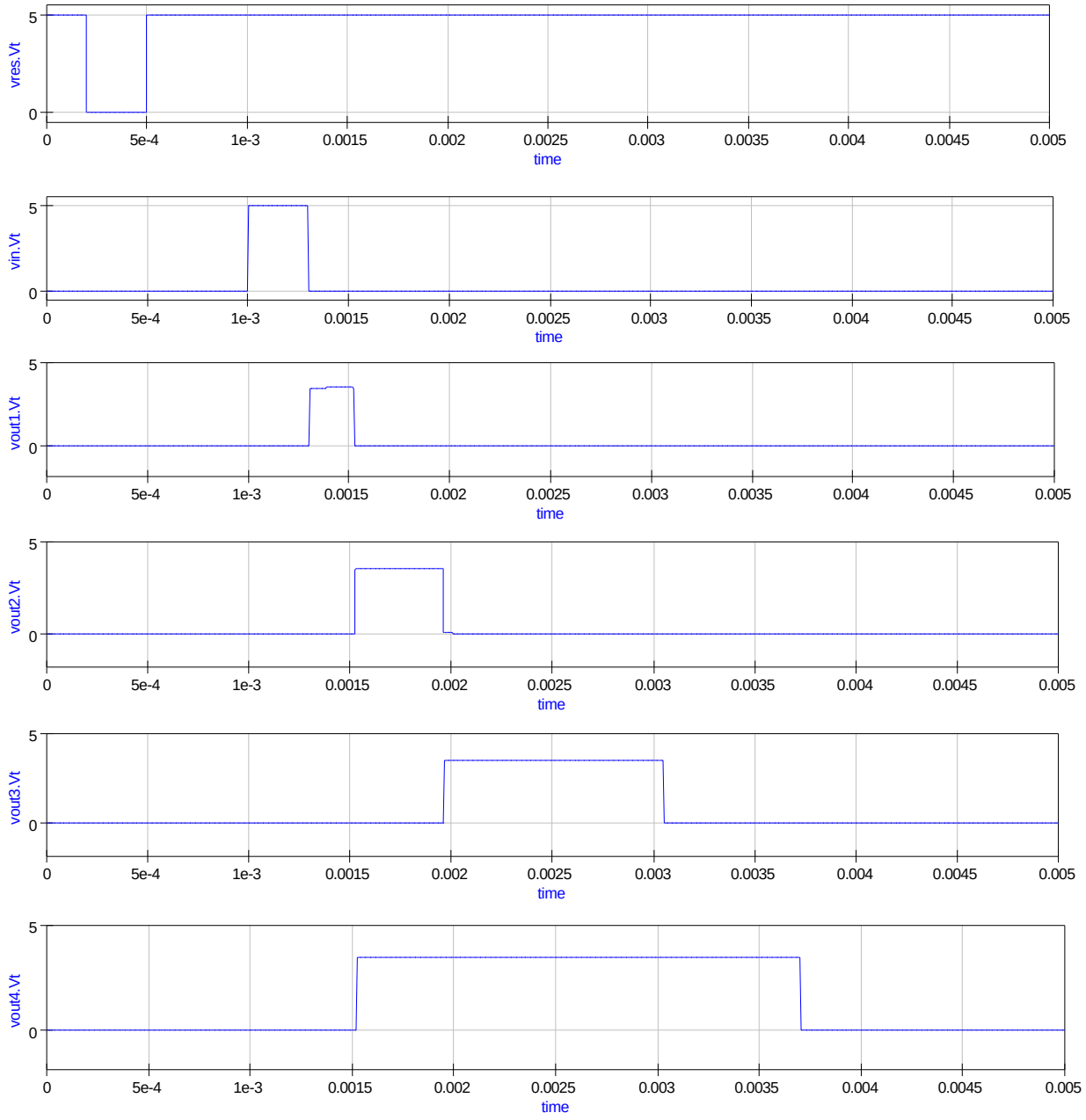


Figure 8.19: Simulation waveforms for the monostable pulse generator circuit.

8.4.2 Frequency divider circuit

A common requirement in both digital and mixed mode circuit design is frequency division, where a high frequency pulse train, often derived from a crystal controlled clock, is divided down to a much lower frequency[17]. The classical way of dividing such signals is to use a chain of flip-flops each connected as a divide by two element. The 555 timer can also be used for pulse train frequency division[18]. The schematic shown in Fig. 8.20 shows a basic monostable mode 555 circuit with a train of pulses applied to the 555 trigger input pin 2 (TRIG). In an earlier section of these notes it was explained that the 555 trigger comparator input was signal level sensitive and retriggering takes place if the duration of the low signal section of the trigger waveform is greater than the monostable pulse duration. In Fig. 8.20 the monostable pulse length is 0.22ms and rectangular voltage generator parameter TL is 0.5ms which causes retriggering to occur. The effects of retriggering can be seen in Fig. 8.21. Frequency division employing 555 timers is based on the monostable circuit shown in Fig. 8.20 and hence circuit designers must make sure that retriggering does not take place. Illustrated in Fig. 8.22 is a two stage frequency division circuit where each stage divides the input pulse train by five giving an overall division ratio of twenty five. The output waveforms for this circuit are shown in Fig. 8.23. When designing 555 timer frequency divider circuits good performance can be achieved if the period of the 555 timer is set at (N-0.5) times the period of the input pulse train[19], where N is the division ratio and is in the range $2 \leq N \leq 10$.

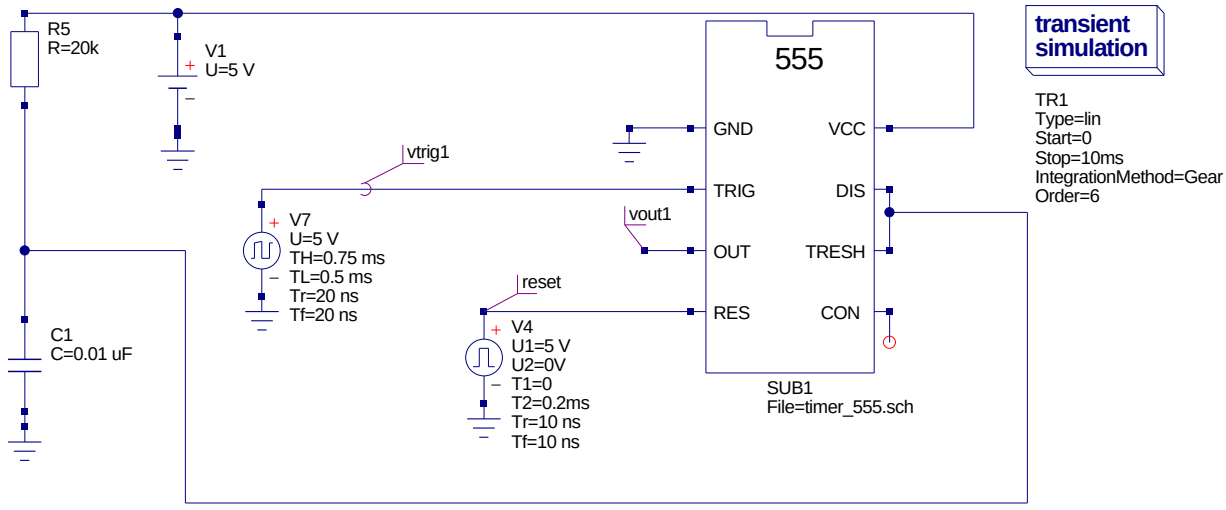


Figure 8.20: A monostable mode 555 timer circuit with a pulse train applied to the trigger input.

[17]Often the resulting frequency is in the region 1 to 5 Hz and is used to flash an LED, or some other optical actuator, on/off.

[18]555 timers are normally more efficient than flip-flops in this application because single devices can have divisors greater than two.

[19]E. A Parr, IC 555 Projects, Bernard Babani (publishing) Ltd, 1981, p. 109.

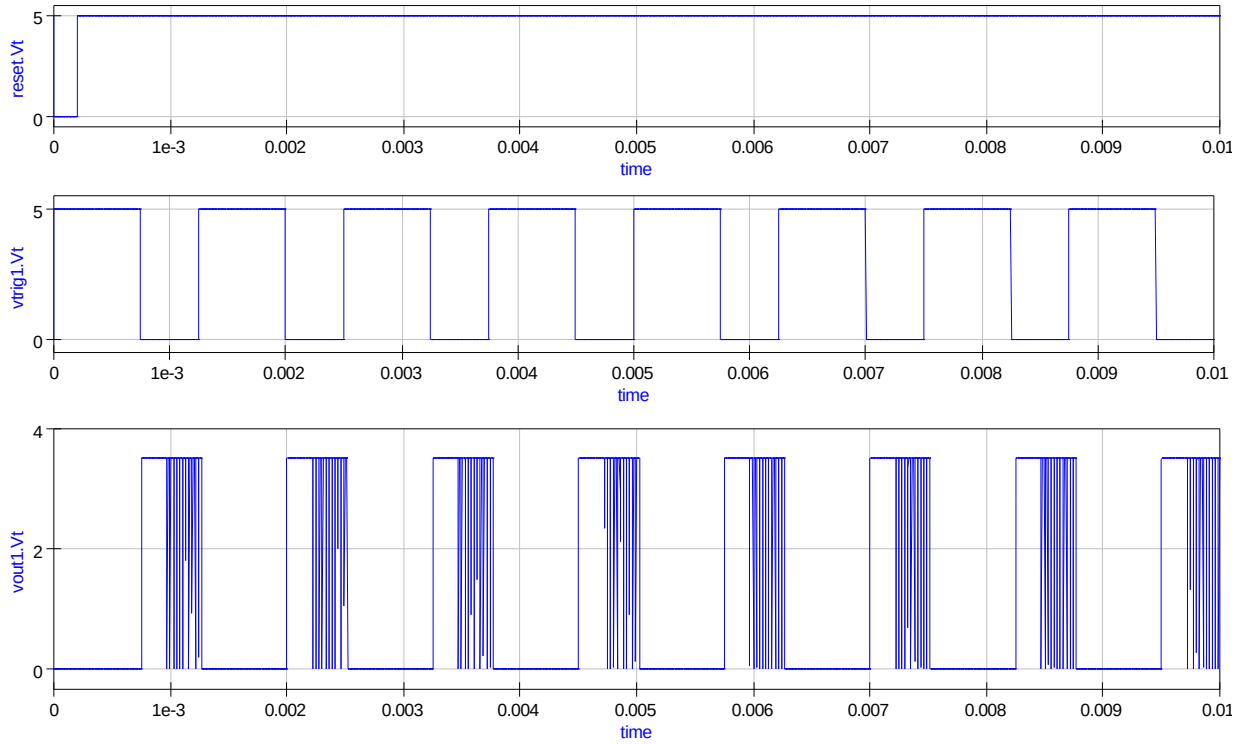


Figure 8.21: Simulation waveforms for the circuit given in Fig. 8.20: these show 555 retriggering.

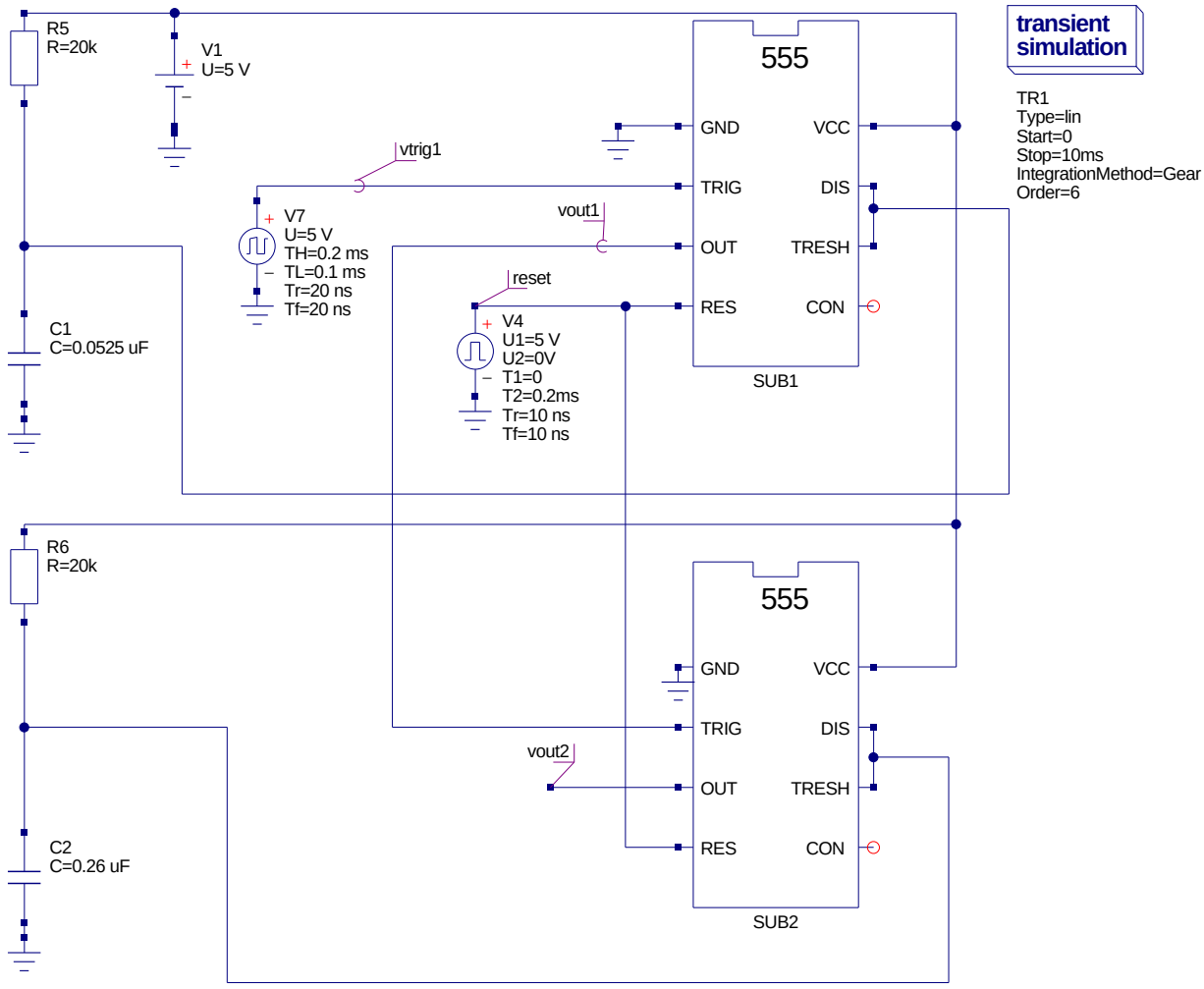


Figure 8.22: A two stage 555 timer frequency division circuit.

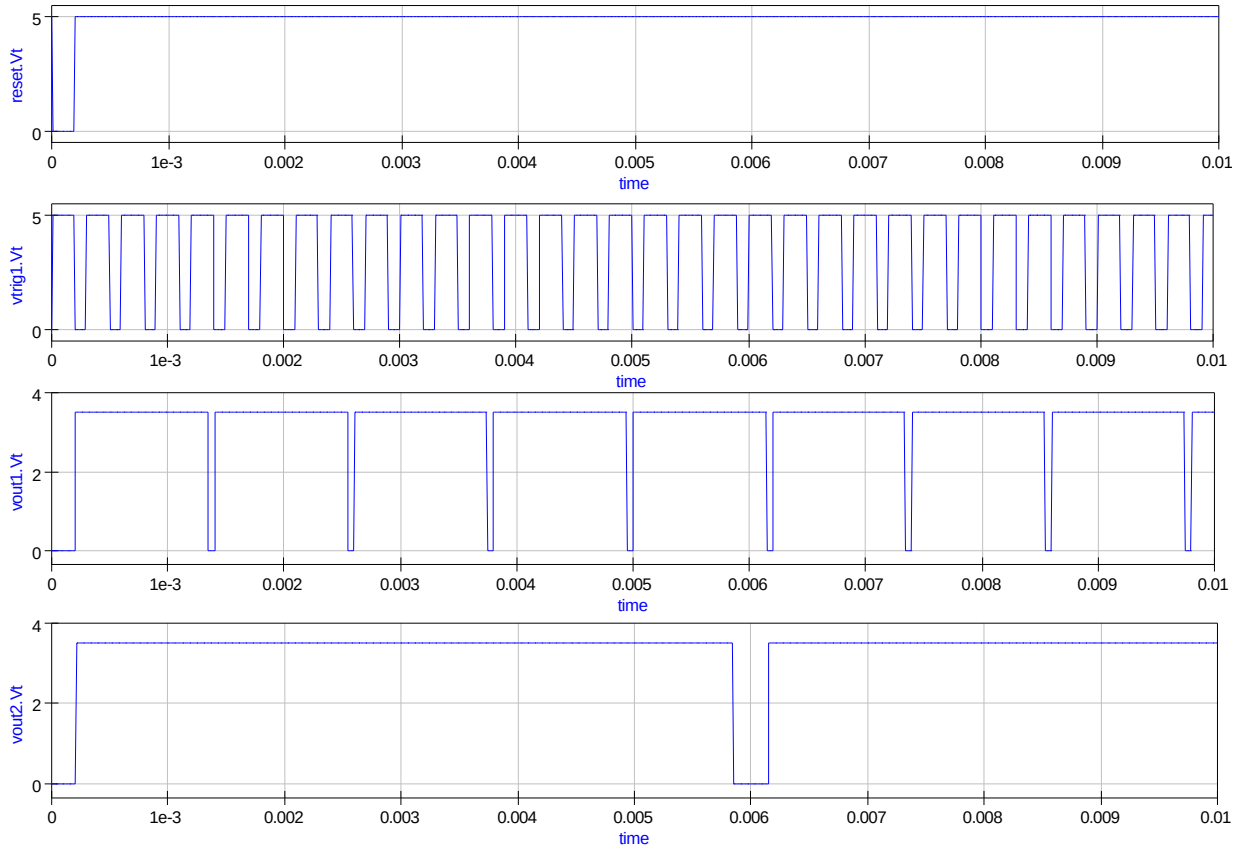


Figure 8.23: Simulation waveforms for the circuit given in Fig. 8.22.

8.5 End note

Developing a simulation model for the 555 timer is an interesting challenge. This tutorial note attempts to describe the principles and macromodelling technology needed for such a task. It also demonstrates how much Qucs has matured as a universal simulator. The new Qucs 555 timer model is very much a first attempt on my part at building a functional model of this complex device. Much more work needs to be done in the future to improve the 555 timer model. Low power 555 timer models are also needed for these popular variants. Longer term a universal parameterised subcircuit model for the 555 timer should become possible once passing parameters to Qucs subcircuits and calculation of component values using equations are implemented. A special thanks to Stefan Jahn for all his encouragement and the many modifications he made to Qucs, which either corrected bugs or added functionality, during the period I have been working on this topic.

9 Qucs Simulation of SPICE Netlists

9.1 Introduction

During the 1960's and 70's, the academic community worked tirelessly to develop computer simulation programs that could act as aids in the process of circuit design. One of the best known of these programs is SPICE[1]. First released in 1972 by the University of California at Berkeley, SPICE has become an industrial standard circuit simulator. Qucs is a modern circuit simulation program which attempts to bring together a range of established and emerging circuit simulation technologies to form a "Quite Universal Circuit Simulator". Although not yet finished, a substantial part of the central core of the package is functioning, allowing it to be used as a simulation engine for the analysis and design of real circuits. Many of the basic circuit components and simulation domains found in SPICE are also available in Qucs. Over the last three decades the SPICE simulation circuit netlist language has become a standard for describing, interchanging and publishing semiconductor device models and circuit data. Today, most semiconductor device manufacturers provide SPICE models or subcircuit netlists for their discreet components and integrated circuits. One area where Qucs and SPICE differ significantly is in their circuit file netlist formats which are very different[2]. Qucs cannot directly simulate standard SPICE circuit netlists but requires them to be converted to their Qucs equivalent prior to simulation. The purpose of this tutorial note is to introduce readers to a number of techniques that allow SPICE netlists to be simulated by Qucs, secondly to indicate the limitations of the current SPICE to Qucs netlist conversion process, and finally to present a preview of how Qucs is likely develop in the future in the area of SPICE netlist compatibility.

9.2 The basic SPICE netlist format

SPICE simulation input data are text files which describe circuit structure, component data and requested simulation tasks for the circuit who's performance is being simulated. Such text files form the fundamental input data to the SPICE simulation engine, and normally include:

[1] The origins and background to the development of the SPICE simulator are described by Ronald A. Rohrer in Circuit Simulation - the early years, illuminating SPICE's strengths, uncovering weaknesses, and projecting its future, IEEE Circuits and Devices, 1992, pp 32-37.

[2] The Qucs netlist grammar is defined in appendix A1, of the Qucs Technical Papers.

- A title statement
- Circuit node names
- Circuit element values
- Voltage and current source descriptions
- Analysis command statements
- Output data statements
- Other command statements

In SPICE 2[3]circuit node names (nets) are identified by integers numbered from 0 to 9999. SPICE 3[4] allows a mixture of letters and numbers for node names. All circuit nodes must have a DC path to ground. Ground node is always node 0 and is considered global. Circuit element values are expressed as integers or real numbers in scientific notation, for example 5, 0.5e1 5.0, or in engineering notation using suffixes. The available SPICE suffixes are f = 1e-15 (femto), p = 1e-12 (pico), n = 1e-9 (nano), u = 1e-6 (micro), mil = 25e-6, m = 1e-3 (milli), k = 1e3 (kilo), meg = 1e6 (mega), g = 1e9 (giga) and t = 1e12 (tera). Component unit abbreviations are allowed in circuit value descriptions. However, these must not be separated from their associated values by spaces. Commonly used unit abbreviations are V = Volt, A = Amps. Hz = Hertz, ohm = Ohm(Ω), H = Henry, F = Farad and deg = Degree. SPICE input data files have the following format:

1. Title
2. * starts a comment line
3. Circuit description
4. Simulation directives
5. Data output directives
6. .end

[3]A guide to SPICE 2 features and simulation data format is given in SPICE Version 2G User's Guide, A Vladimirescu, Kaihe Zhang, A.R. Newton, D. O. Pederson and A. Sangiovanni-Vincentelli, August 1981, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Ca., 94720, US.

[4]See SPICE 3 Version 3F User's Manual, B. Johnson, T. Quarles,A.R. Newton, D. O. Pederson and A. Sangiovanni-Vincentelli, October 1992, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Ca., 94720, US.

A typical SPICE input data file for a discreet component circuit is shown in Fig. 9.1. In this netlist all nodes are shown numbered, following the SPICE 2 node naming convention. Also the power supply, AC input signal generator and output load are not included. Essentially, the netlist shown in Fig. 9.1 represents the amplifier without any external components connected to it. Although Qucs cannot directly simulate SPICE netlists the software does contain a SPICE to Qucs netlist conversion program called QUCSCONV. This routine takes as input a SPICE netlist file and outputs an equivalent Qucs formatted netlist file. The Qucs netlist file can be read and simulated by the Qucs simulation engine. To make the process transparent, and indeed straightforward for users, the conversion stage in simulating SPICE netlist files[5] has been automated via the Qucs GUI simulate command (F2 key). SPICE netlist files can be linked to a Qucs SPICE netlist schematic symbol.[6] These in turn can be connected, on a schematic, to any other appropriate Qucs component symbol or user defined symbol. Figure 9.2 shows the resulting schematic for the two stage BJT circuit. In this diagram the external voltage sources and amplifier load have been added together with the usual Qucs icons for DC and AC simulation of the circuit. During simulation Qucs treats the SPICE netlist component as a subcircuit[7] and generates the appropriate Qucs netlist code. For example, the netlist shown in Fig. 9.3 illustrates the Qucs style netlist code for the two stage BJT amplifier. Simulation of the two stage BJT amplifier gives the output waveforms displayed in Fig. 9.4.

[5]For convenience SPICE netlist files are often denoted with the extention cir and stored in a Qucs project under the *other* category.

[6]The schematic symbol SPICE netlist can be found in the file components section of the components icon lists on the left hand side of the GUI. Its connection pin list may be setup and edited via the Edit SPICE component properties dialogue.

[7]Hence the need to separate the external voltage sources and amplifier load from the main amplifier circuit.

```
* A two-stage BJT amplifier.
*
* Input node 2, output node 9
* Power supply Vcc connected to node 10
*
c1 2 3 10uf
r1 3 10 200k
r2 3 0 50k
r5 10 4 12k
q1 4 3 5 qmod
r6 5 0 3.6k
c2 4 6 10uf
c4 5 0 15uf
r3 10 6 120k
r4 6 0 30k
r7 10 7 6.8k
q2 7 6 8 qmod
r8 8 0 3.6k
c5 8 0 25uf
c3 7 9 10uf
*
.model qmod npn (is=2e-16 bf=50 br=1 rb=5 rc=1 re=0
+ cje=0.4pf vje=0.8 me=0.4 cjc=0.5pf vjc=0.8 ccs=1pf va=100)
*
.end
```

Figure 9.1: SPICE netlist for a simple two stage BJT amplifier.

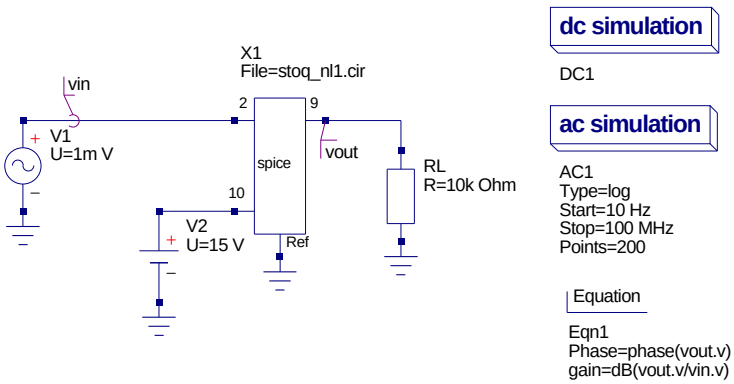


Figure 9.2: Qucs schematic for the two stage amplifier represented by the SPICE netlist shown in Fig. 9.1.

```
.Def:stoq_nl1_cir _net2 _net9 _net10 _ref
  C:C3 _net7 _net9 C="10uF"
  C:C5 _net8 _ref C="25uF"
  R:R8 _net8 _ref R="3.6k"
  BJT:Q2 _net6 _net7 _net8 _ref Type="npn" Is="2e-16" Bf="50" Br="1"
       Rb="5" Rc="1" Re="0" Cje="0.4pF"Vje="0.8" Mje="0.4" Cjc="0.5pF"
       Vjc="0.8" Cjs="1pF" Vaf="100" Nf="1" Nr="1" Ikf="0" Ikr="0" Var="0"
       Ise="0" Ne="1.5" Isc="0" Nc="2" Rbm="0" Irb="0" Mjc="0.33" Xcjc="1"
       Vjs="0.75" Mjs="0" Fc="0.5" Vtf="0" Tf="0" Xtf="0" Itf="0" Tr="0"
  R:R7 _net10 _net7 R="6.8k"
  R:R4 _net6 _ref R="30k"
  R:R3 _net10 _net6 R="120k"
  C:C4 _net5 _ref C="15uF"
  C:C2 _net4 _net6 C="10uF"
  R:R6 _net5 _ref R="3.6k"
  BJT:Q1 _net3 _net4 _net5 _ref Type="npn" Is="2e-16" Bf="50" Br="1"
       Rb="5" Rc="1" Re="0" Cje="0.4pF"Vje="0.8" Mje="0.4" Cjc="0.5pF"
       Vjc="0.8" Cjs="1pF" Vaf="100" Nf="1" Nr="1" Ikf="0" Ikr="0" Var="0"
       Ise="0" Ne="1.5" Isc="0" Nc="2" Rbm="0" Irb="0" Mjc="0.33" Xcjc="1"
       Vjs="0.75" Mjs="0" Fc="0.5" Vtf="0" Tf="0" Xtf="0" Itf="0" Tr="0"
  R:R5 _net10 _net4 R="12k"
  R:R2 _net3 _ref R="50k"
  R:R1 _net3 _net10 R="200k"
  C:C1 _net2 _net3 C="10uF"
.Def:End
```

Figure 9.3: Qucs format netlist for the two stage BJT amplifier: NOTE -In this listing the entries for Q1 and Q2 have been edited so that they fit on the text page.

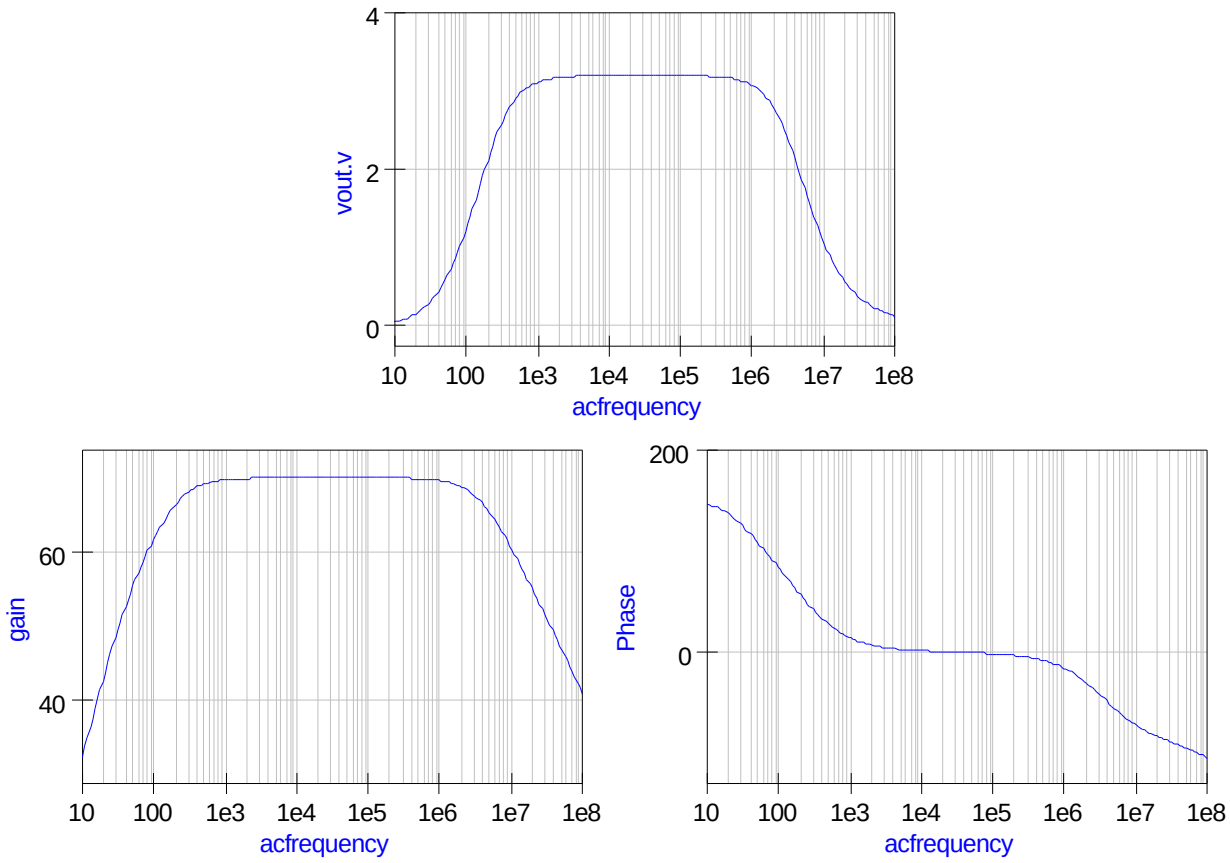


Figure 9.4: Simulation waveforms for the two stage amplifier.

9.3 Defining symbols for Qucs SPICE netlist components

Qucs automatically generates the symbol for a SPICE netlist component and does not allow users to edit the resulting symbol. One of the disadvantage of this feature is that the placement of the symbol input and output pins may be in a position which is contrary to accepted use or signal flow direction. To overcome this limitation a user defined symbol may be constructed where the SPICE netlist component is embedded within the new symbol. Figure 9.5 illustrates such a symbol for the two stage BJT amplifier and the resulting Qucs netlist for the new symbol is shown in Fig. 9.6. From Fig. 9.6 we observe that embedding a SPICE netlist symbol, within a user defined symbol, introduces an additional subcircuit call in the resulting Qucs netlist; this is probably a small price to pay for the convenience that a user defined symbol brings to the overall simulation process.

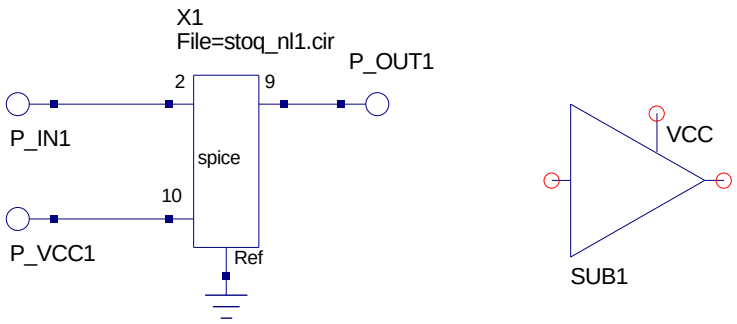


Figure 9.5: User defined symbol for the two stage BJT amplifier.

```
.Def:stoq_fig5_amp _net0 _net1 _net2
Sub:X1 _net0 _net1 _net2 gnd Type="stoq_nl1_cir"
.Def:End

.Def:stoq_nl1_cir _net2 _net9 _net10 _ref
  C:C3 _net7 _net9 C="10uF"
  C:C5 _net8 _ref C="25uF"
  R:R8 _net8 _ref R="3.6k"
  BJT:Q2 _net6 _net7 _net8 _ref Type="npn" Is="2e-16" Bf="50" Br="1"
       Rb="5" Rc="1" Re="0" Cje="0.4pF"Vje="0.8" Mje="0.4" Cjc="0.5pF"
       Vjc="0.8" Cjs="1pF" Vaf="100" Nf="1" Nr="1" Ikf="0" Ikr="0" Var="0"
       Ise="0" Ne="1.5" Isc="0" Nc="2" Rbm="0" Irb="0" Mjc="0.33" Xcjc="1"
       Vjs="0.75" Mjs="0" Fc="0.5" Vtf="0" Tf="0" Xtf="0" Itf="0" Tr="0"
  R:R7 _net10 _net7 R="6.8k"
  R:R4 _net6 _ref R="30k"
  R:R3 _net10 _net6 R="120k"
  C:C4 _net5 _ref C="15uF"
  C:C2 _net4 _net6 C="10uF"
  R:R6 _net5 _ref R="3.6k"
 BJT:Q1 _net3 _net4 _net5 _ref Type="npn" Is="2e-16" Bf="50" Br="1"
       Rb="5" Rc="1" Re="0" Cje="0.4pF"Vje="0.8" Mje="0.4" Cjc="0.5pF"
       Vjc="0.8" Cjs="1pF" Vaf="100" Nf="1" Nr="1" Ikf="0" Ikr="0" Var="0"
       Ise="0" Ne="1.5" Isc="0" Nc="2" Rbm="0" Irb="0" Mjc="0.33" Xcjc="1"
       Vjs="0.75" Mjs="0" Fc="0.5" Vtf="0" Tf="0" Xtf="0" Itf="0" Tr="0"
  R:R5 _net10 _net4 R="12k"
  R:R2 _net3 _ref R="50k"
  R:R1 _net3 _net10 R="200k"
  C:C1 _net2 _net3 C="10uF"
.Def:End
```

Figure 9.6: Qucs format netlist for the two stage BJT amplifier represented by a user defined symbol: NOTE -In this listing the entries for Q1 and Q2 have been edited so that they fit on the text page.

9.4 Handling SPICE subcircuits

Although Qucs treats SPICE netlist components as subcircuits the SPICE to Qucs netlist conversion process still allows SPICE subcircuits to be defined within the SPICE file being converted. Such subcircuits then become local subcircuits to the SPICE netlist component to which they are attached. This allows complex circuits consisting of many related, but often different, circuit blocks to be represented by a single symbol in a Qucs schematic. In such cases the resulting symbol represents a true subsection of an entire circuit rather than a simple single circuit function subcircuit. To demonstrate this feature consider the following examples; (1) a multisection LC delay line and (2) a CMOS ring counter.

9.4.1 Subcircuit example 1: a multisection LC delay line

The SPICE netlist for a ten section LC passive delay line is shown in Fig. 9.7. In this listing each LC delay section is represented by a SPICE subcircuit and these sections are connected in series to form the overall delay line. Figures 9.8 and 9.9 present the resulting Qucs netlist and generated waveforms obtained with the test circuit shown in Fig. 9.10.

9.4.2 Subcircuit example 2: a two section CMOS ring counter

Subcircuit example one only contains a single local subcircuit. The next example demonstrates how SPICE listings with more than one subcircuit are handled by Qucs. Such circuits are representative of more complex electronic systems which form easily identifiable subsystem blocks.[8] Fig. 9.11 shows the SPICE netlist for a simple two section CMOS ring counter. This circuit is modelled at discreet component level and uses basic level one MOS parameters to define the MOS transistors. These are then combined to form NAND and NOR subcircuits. Again for completeness the resulting Qucs netlist is shown in Fig. 9.12 together with a typical set of counter input and output signal waveforms, Fig. 9.13.

[8] One significant advantage that Qucs has when compared to netlist entry only circuit simulators is that it is possible the define schematic symbols for subsystem blocks that comprise discreet components and one or more local subcircuits. These may then be employed like any other Qucs symbols when constructing circuit schematics.

```
* Z0 = 320 Ohm.
*
.subckt lc n1 n2
l1 n1 n2 10uh
c1 n2 0 10pf
.ends
*
rs n9 n10 320ohm
x1 n10 n11 lc
x2 n11 n12 lc
x3 n12 n13 lc
x4 n13 n14 lc
x5 n14 n15 lc
x6 n15 n16 lc
x7 n16 n17 lc
x8 n17 n18 lc
x9 n18 n19 lc
x10 n19 n20 lc
rl n20 0 320ohm
.end
```

Figure 9.7: SPICE netlist for a ten section LC delay line..

```
.Def:stoq_fig10a _net0 _net10 _net1 _net2 _net3 _net4
                 _net5 _net6 _net7 _net8 _net9
Sub:X1 _net0 _net10 _net1 _net2 _net3 _net4
            _net5 _net6 _net7 _net8 _net9 gnd Type="test3_pp_cir"
.Def:End

.Def:test3_pp_cir _netN9 _netN11 _netN12 _netN13 _netN14
                  _netN15 _netN16 _netN17 _netN18 _netN19 _netN20 _ref
  R:RL _netN20 _ref R="320Ohm"
  Sub:X10 _ref _netN19 _netN20 Type="LC"
  Sub:X9 _ref _netN18 _netN19 Type="LC"
  Sub:X8 _ref _netN17 _netN18 Type="LC"
  Sub:X7 _ref _netN16 _netN17 Type="LC"
  Sub:X6 _ref _netN15 _netN16 Type="LC"
  Sub:X5 _ref _netN14 _netN15 Type="LC"
  Sub:X4 _ref _netN13 _netN14 Type="LC"
  Sub:X3 _ref _netN12 _netN13 Type="LC"
  Sub:X2 _ref _netN11 _netN12 Type="LC"
  Sub:X1 _ref _netN10 _netN11 Type="LC"
  R:RS _netN9 _netN10 R="320Ohm"
  .Def:LC _ref _netN1 _netN2
  L:L1 _netN1 _netN2 L="10uH"
  C:C1 _netN2 _ref C="10pF"
  .Def:End
.Def:End
```

Figure 9.8: Qucs netlist for a 10 section LC delay line: NOTE -In this listing the entries for the .Def statements have been edited so that they fit on the text page.

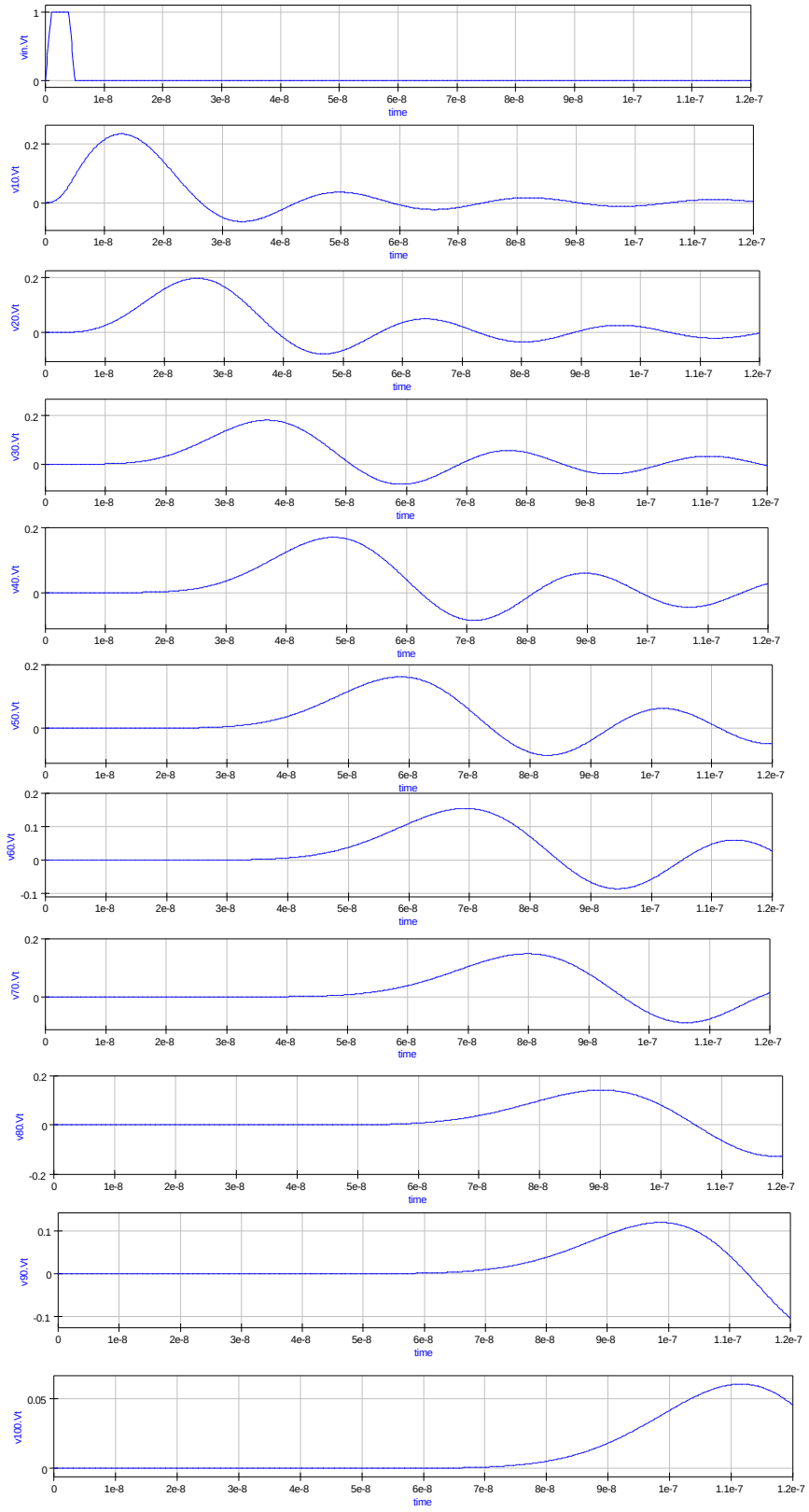


Figure 9.9: Simulation waveforms for a 10 section LC delay line.

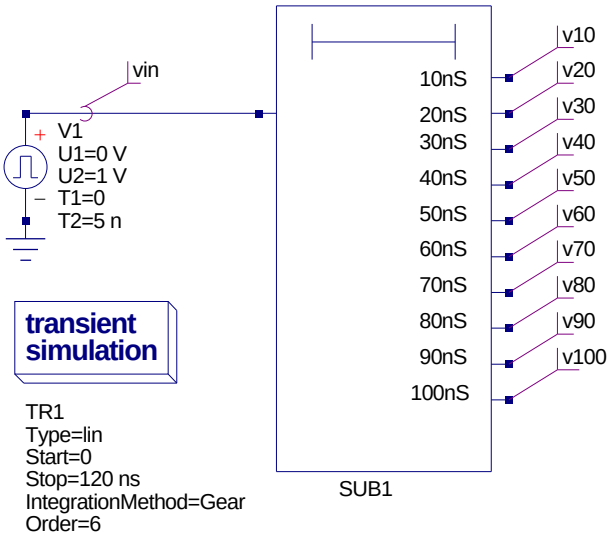


Figure 9.10: LC delay line test circuit.

```
* Two stage CMOS ring counter circuit.
*
x1 1 5 6  nand2
x2 1 6 7  nand2
x3 3 6 2  nand2
x4 2 7 3  nand2
x5 1 2 8  nor2
x6 1 8 9  nor2
x7 5 8 4  nor2
x8 4 9 5  nor2
*
.model modp pmos( vto=-1 kp=10u
+                        cgdo=0.2n cgso=0.2n cgbo=2n)
.model modn nmos(vto=1 kp=10u
+                        cgdo=0.2n cgso=0.2n cgbo=2n)
*
.subckt nand2 1 2 3
m1 3 1 4 4 modp w=40u l=5u
m2 3 2 4 4 modp w=40u l=5u
m3 5 1 0 0 modn w=20u l=5u
m4 3 2 5 5 modn w=20u l=5u
c1 1 0 10p
c2 2 0 10p
vcc 4 0 pulse ( 0 5 0 1ns 1ns 1 2)
.ends
*
.subckt nor2 1 2 3
m1 4 1 7 7 modp w=40u l=5u
m2 3 2 4 4 modp w=40u l=5u
m3 3 2 0 0 modn w=20u l=5u
m4 3 1 0 0 modn w=20u l=5u
c1 1 0 10p
c2 2 0 10p
vcc 7 0 pulse ( 0 5 0 1ns 1ns 1 2)
.ends
.end
```

Figure 9.11: SPICE netlist for a two section CMOS ring counter.

```
# Qucs 0.0.11   /media/hda2/OPAMP_templates/test_stoq_fig11a.sch
.Def:stoq_fig11a_cir _net1 _net4 _ref
  .Def:NOR2 _ref _net1 _net2 _net3
  Vpulse:VCC _net7 _cnet0 U1="0" U2="5" T1="0" Tr="1ns" Tf="1ns" T2="1"
  MOSFET:M1 _net1 _net4 _net7 _net7 Type="pfet" W="40u" L="5u" Vt0="-1"
        Kp="10u" Cgdo="0.2n" Cgso="0.2n" Cgbo="2n" Is="1e-14" N="1"
        Lambda="0" Gamma="0" Phi="0.6"
  MOSFET:M2 _net2 _net3 _net4 _net4 Type="pfet" W="40u" L="5u" Vt0="-1"
        Kp="10u" Cgdo="0.2n" Cgso="0.2n" Cgbo="2n" Is="1e-14" N="1"
        Lambda="0" Gamma="0" Phi="0.6"
  MOSFET:M3 _net2 _net3 _ref _ref Type="nfet" W="20u" L="5u" Vt0="1"
        Kp="10u" Cgdo="0.2n" Cgso="0.2n" Cgbo="2n" Is="1e-14" N="1"
        Lambda="0" Gamma="0" Phi="0.6"
  MOSFET:M4 _net1 _net3 _ref _ref Type="nfet" W="20u" L="5u" Vt0="1"
        Kp="10u" Cgdo="0.2n" Cgso="0.2n" Cgbo="2n" Is="1e-14" N="1"
        Lambda="0" Gamma="0" Phi="0.6"
  C:C1 _net1 _ref C="10p"
  C:C2 _net2 _ref C="10p"
  Vdc:VCC _cnet0 _ref U="0"
  .Def:End
  .Def:NAND2 _ref _net1 _net2 _net3
  Vpulse:VCC _net4 _cnet1 U1="0" U2="5" T1="0" Tr="1ns" Tf="1ns" T2="1"
  MOSFET:M1 _net1 _net3 _net4 _net4 Type="pfet" W="40u" L="5u" Vt0="-1"
        Kp="10u" Cgdo="0.2n" Cgso="0.2n" Cgbo="2n" Is="1e-14" N="1"
        Lambda="0" Gamma="0" Phi="0.6"
  MOSFET:M2 _net2 _net3 _net4 _net4 Type="pfet" W="40u" L="5u" Vt0="-1"
        Kp="10u" Cgdo="0.2n" Cgso="0.2n" Cgbo="2n" Is="1e-14" N="1"
        Lambda="0" Gamma="0" Phi="0.6"
  MOSFET:M3 _net1 _net5 _ref _ref Type="nfet" W="20u" L="5u" Vt0="1"
        Kp="10u" Cgdo="0.2n" Cgso="0.2n" Cgbo="2n" Is="1e-14" N="1"
        Lambda="0" Gamma="0" Phi="0.6"
  MOSFET:M4 _net2 _net3 _net5 _net5 Type="nfet" W="20u" L="5u" Vt0="1"
        Kp="10u" Cgdo="0.2n" Cgso="0.2n" Cgbo="2n" Is="1e-14" N="1"
        Lambda="0" Gamma="0" Phi="0.6"
  C:C1 _net1 _ref C="10p"
  C:C2 _net2 _ref C="10p"
  Vdc:VCC _cnet1 _ref U="0"
  .Def:End
  Sub:X8 _ref _net4 _net9 _net5 Type="NOR2"
  Sub:X7 _ref _net5 _net8 _net4 Type="NOR2"
  Sub:X6 _ref _net1 _net8 _net9 Type="NOR2"
  Sub:X5 _ref _net1 _net2 _net8 Type="NOR2"
  Sub:X4 _ref _net2 _net7 _net3 Type="NAND2"
  Sub:X3 _ref _net3 _net6 _net2 Type="NAND2"
  Sub:X2 _ref _net1 _net6 _net7 Type="NAND2"
  Sub:X1 _ref _net1 _net5 _net6 Type="NAND2"
.Def:End
Sub:X1 vin vout gnd Type="stoq_fig11a_cir"
Vrect:V1 vin gnd U="5 V" TH="1 us" TL="1 us" Tr="1 ns" Tf="1 ns" Td="0 ns"
.TR:TR1 Type="lin" Start="0" Stop="30u" Points="1000" IntegrationMethod="Trapezoidal"
Order="2" InitialStep="0.01 ns" MinStep="1e-18" MaxIter="150" reltol="0.01"
abstol="1 uA" vntol="100 uV" Temp="26.85" LTEreltol="1e-3" LTEabstol="1e-4"
LTEfactor="1" Solver="CroutLU" relaxTSR="no" initialDC="yes" MaxStep="0"
```

Figure 9.12: Qucs netlist for a two section CMOS ring counter: NOTE -In this listing the entries for MOSFETs and transient analysis have been edited so that they fit on the text page.

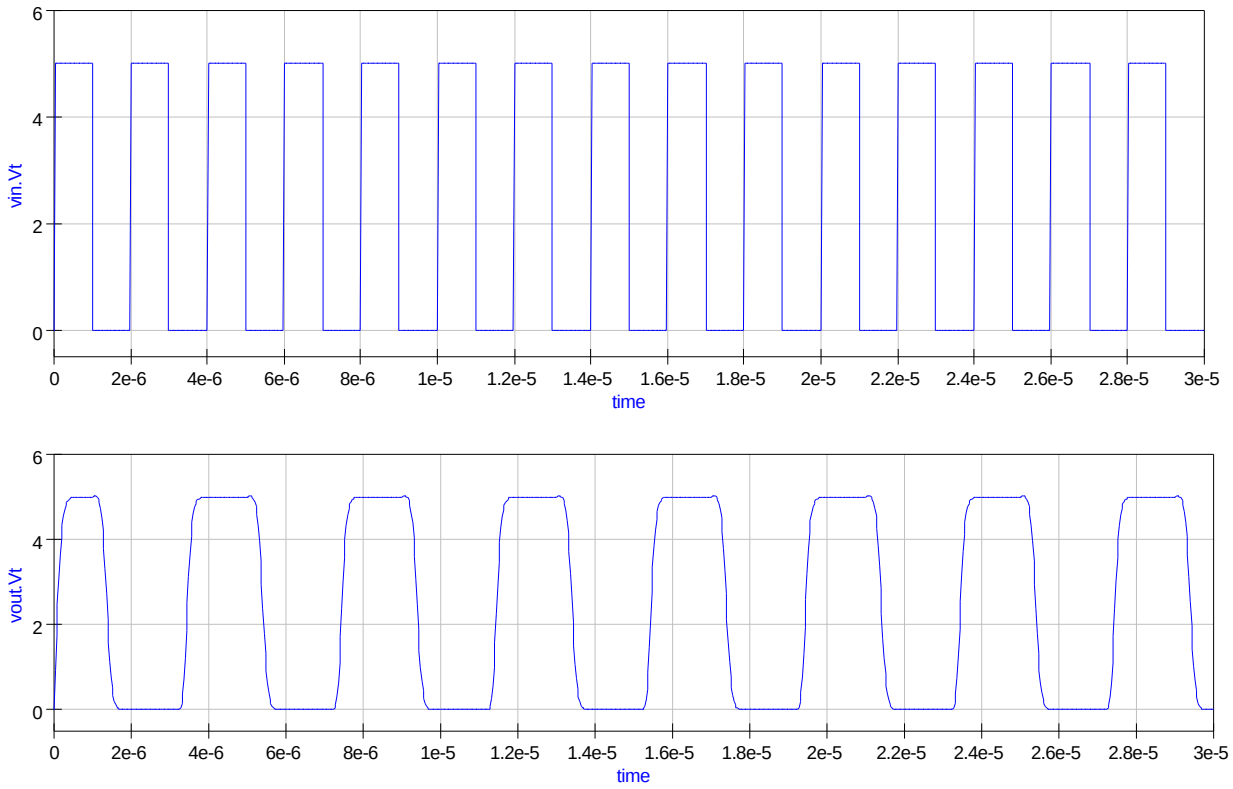


Figure 9.13: Two stage CMOS ring counter signal waveforms.

9.5 Limitations when converting SPICE netlists

Not all SPICE netlists can be converted to Qucs netlist format and simulated by Qucs[9]. There are a number of reasons for this. The first and most obvious is due to the fact that some SPICE components have not been implemented in Qucs yet. Nonlinear controlled voltage and current sources are an example.[10] There are also a number of detailed differences between the SPICE and Qucs implementation of components common to both simulators, one being the lack of PWL features in the Qucs independent voltage and current sources. A second area that represents a significant limitation, for those readers who regularly write SPICE netlists as part of their simulation work, is the fact that Qucs contains a much greater range of predefined primitive components that are not available in either the SPICE 2 or SPICE 3 simulators. Perhaps this is not so much a limitation but an indication of the current development effort being put into Qucs by the development team. As the development of Qucs progresses it is expected that all the component features found in SPICE will have a corresponding entry in Qucs[11].

9.6 Extending the SPICE netlist language

The standard SPICE 2 and SPICE 3 hardware description languages do not allow (1) component values to be defined by algebraic equations[12] or (2) parameters to be passed to subcircuits. This makes writing universal subcircuit models very difficult, forcing semiconductor device manufacturers to issue individual SPICE models for each device they manufacture rather than a single generalised model[13] for a given type of integrated circuit. A well known example being the SPICE Boyle[14] operational amplifier models. A number of current commercial circuit simulators[15] have been extended to include the parameter based features outlined above. In the case of those simulators based on the unextended Berkely SPICE 2G6 or SPICE 3F5[16] code a different approach is often adopted. This is

[9] A number of Qucs users have reported problems in the past when trying to simulate SPICE netlists for components that have been published by device manufactures, see for example, "Qucs SPICE error - please...", William Flyn <WF215@ca...>, 29.8.2006, Qucs help forum.

[10] SPICE 2 polynomal controlled voltage and current sources and SPICE 3 type B sources are not implemented in any of the Qucs versions so far released. Their implementation is on the to-do list but no date for their implementation has been fixed yet.

[11] Future plans in this area are discussed in a later section of these notes.

[12] Please note this is not strictly true as SPICE 3 B sources can be defined by equations involving simulation variables and other data.

[13] In a generalised model only one model description is provided for each generic component/circuit. Different component models are formed by passing parameters to the generalised model. SPICE employs this approach to represent semiconductor devices through the use of the .model statement. However, in the .model case the code for each type of semiconductor device is hardwired into the simulator code rather than being defined by a subcircuit.

[14] Boyle,G.R., B.M. Cohn, D.O. Pederson, and J.E. Solomon, 1974, Macromodeling of integrated circuit amplifiers, IEEE Journal of Solid-State Circuits (December).

[15] For example PSPICE, HSPICE and IS-SPICE.

[16] For example NGSPICE, TCLSPICE and WINSPICE.

based on the use of a preprocessor, similar to that found in the C language, which takes as input a parameter and equation style netlist and outputs a standard SPICE netlist with the parameters and equations evaluated to give a numerical result. The advantage of this approach is that the preprocessor can be used with any SPICE simulator or indeed with Qucs. Two such preprocessors are SPICEPRM and SPICEPP.[17] The flow diagram for the Qucs simulation sequence including a SPICE preprocessing stage is shown in Fig. 9.14. This diagram clearly shows how both standard SPICE and parameterised netlists can be linked into the Qucs simulation cycle. Of the two SPICE preprocessors introduced above SPICEPP is probably the most useful from a Qucs users point of view[18] as it adds more features to the overall simulation process. Hence the notes that follow will concentrate on describing how SPICEPP can be used with Qucs.

9.6.1 The SPICEPP preprocessor

SPICEPP[19] is a preprocessor for Berkeley SPICE 3F5, adding support for a number of structures found in commercial SPICE simulators, specifically SPICE commands .param, .global, .lib, .temp, .meas and inline comments (`$`). The remainder of these notes explain the use of commands .param, .global and the inline comment as these add specific functionality to Qucs that is not provided by other sections of the Qucs simulation software. The definition of these commands are:

- .param data=dataval <data2=dataval2> ..... The .param statement adds the ability to parameterise SPICE data, including component values, voltages, currents and equations.
- .globel node1 <node2> ..... The .global statement causes the named nodes to override local subcircuit nodes of the same name.
- Algebraic statements are enclosed in quotes ‘ ‘[20].
- Inline comments start with the `$` symbol and continue to the end of a line.

[17](1) Andrew J. Borsa, SPICEPRM, A SPICE preprocessor for parameterised subcircuits, V 0.11, 1996, <andy@moose.mv.com> (SPICEPRM can be downloaded from the Sourceforge.net ngspice project.) and (2) John Shaehen, SPICEPP, A SPICE proprocessor for SPICE 3F5, V 1.5, 2000, <john@reptechnic.com.au>. (SPICEPP can be downloaded from the Sourceforge.net tclspice project.)

[18]SPICEPP was written after SPICEPRM and extends the facilities offered by SPICEPRM.

[19]SPICEPP is written in PERL. The SPICEPP.pl script should be copied to a directory on your search path. On my system I keep it in the Qucs bin directory. PERL must also be installed on your system.

[20]The ‘ character can be found on the most left key on the row of numerical keys (‘ 1 2 3 4 5 6 7 8 9 0 - .....) - this is the case on my keyboard.

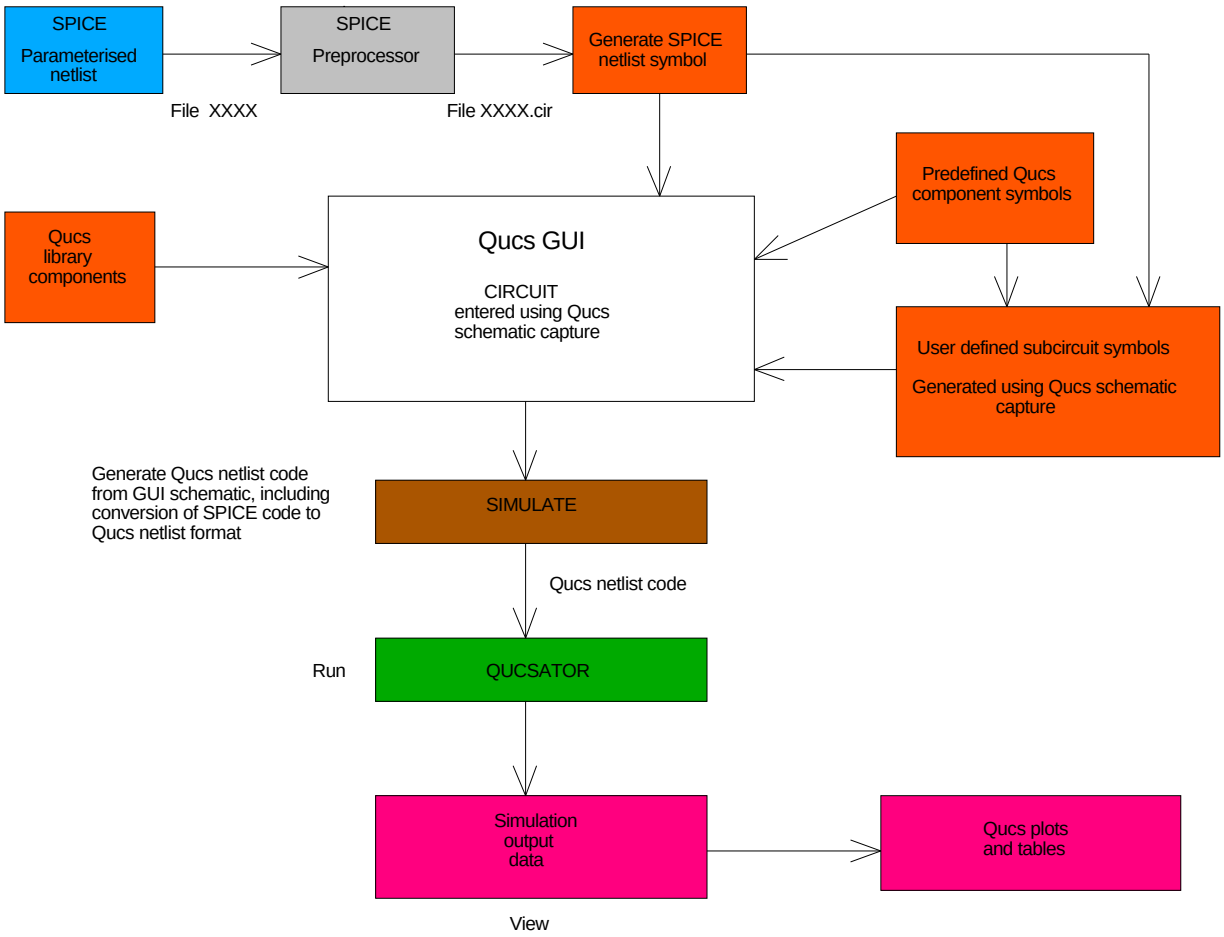


Figure 9.14: Flow diagram of Qucs simulator stages including SPICE preprocessing.

9.7 Circuit template models

When modelling devices or circuits for simulation a particularly productive approach is the use of a universal template that can be employed to generate models for devices of the same type but with different characteristics. By simply changing the parameters embedded in a universal template a new device model is generated when the netlist code is passed through the SPICEPP preprocessor. Consider the SPICE template model shown in Fig. 9.15. This represents a simple modular AC macromodel[21] for an OP AMP. OP AMP internal pins are given by integers and external pins by names in SPICE 3 format. The parameters for a UA741 OP AMP are shown listed at the start of the SPICE preprocessor netlist. These are used in the calculation of the component values in later sections of the netlist. In all cases parameters must be defined before they are used in component calculations. Passing this listing through the SPICEPP preprocessor[22] and generating a Qucs user defined symbol for the UA741 OP AMP results in the Qucs netlist and symbol shown in Figures 9.16 and 9.17. An application of the generated UA741 OP AMP model is shown in Fig. 9.18. This circuit is a notch filter. In Fig. 9.18 the band rejection characteristic of the filter are realised by a twin-T RC network. Figure 9.19 shows the simulated small signal transfer characteristics of this filter.

[21] Details of the model derivation can be found in the Qucs Modelling Operational Amplifiers tutorial, Qucs Web site.

[22] The SPICEPP PERL script can be run from a shell using the command ***spicepp.pl name.pp > name.cir***, where name is the name of the file to be processed.

```
*
* Device pins 1. input  in_n, in_p
*             2. output out
*
* ua741 OP AMP parameters
*
.param voff = 0.7m
.param ib = 80n
.param ioff = 20n
.param rd = 2meg
.param cd = 1.4p
.param cmrrdc = 31622.8
.param fcmz = 200.0
.param aoldc = 199526
.param gbp = 1meg
.param fp2 = 3meg
.param ro = 75.0
*
* input stage
*
voff1 in_n 6 'voff/2'
voff2 7 in_p 'voff/2'
ib1   0 6 ib
ib2   7 0 ib
ioff1 7 6 'ioff/2'
r1    6 8 'rd/2'
r2    7 8 'rd/2'
cin1  6 7 cd
*
* common-mode zero stage
*
ecm1 12 0 8 0 '1e6/cmrrdc'
rcm1 12 13 1meg
ccm1 12 13 '1/(2 * 3.1412 * 1e6 * fcmz)'
rcm2 13 0 1
*
* differential and common-mode
* signal summing stage
*
gmsum1 0 14 7 6 1
gmsum2 0 14 13 0 1
rsum1  14 0 1
*
* voltage gain stage 1
*
gmp1 0 9 14 0 1
rado 9 0 aoldc
cp1 9 0 '1/(2 * 3.1412 * gbp)'
*
* voltage gain stage 2
*
gmp2 0 11 9 0 1
rp2  11 0 1
cp2  11 0 '1/(2 * 3.1412 * fp2)'
*
* output stage
*
eos1 10 0 11 0 1
ros1 10 out ro
*
```

Figure 9.15: SPICE template preprocessor netlist for a UA741 AC modular OP AMP model.

```
.Def:stoq_fig17 _net0 _net1 _net2
Sub:X1 _net0 _net1 _net2 gnd Type="stoq_fig15_cir"
.Def:End

.Def:stoq_fig15_cir _netIN_N _netOUT _netIN_P _ref
  R:ROS1 _net10 _netOUT R="75"
  VCVS:EOS1 _net11 _net10 _ref _ref G="1"
  C:CP2 _net11 _ref C="5.30583e-08"
  R:RP2 _net11 _ref R="1"
  VCCS:GMP2 _net9 _ref _net11 _ref G="1"
  C:CP1 _net9 _ref C="1.59175e-07"
  R:RADO _net9 _ref R="199526"
  VCCS:GMP1 _net14 _ref _net9 _ref G="1"
  R:RSUM1 _net14 _ref R="1"
  VCCS:GMSUM2 _net13 _ref _net14 _ref G="1"
  VCCS:GMSUM1 _net7 _ref _net14 _net6 G="1"
  R:RCM2 _net13 _ref R="1"
  C:CCM1 _net12 _net13 C="7.95874e-10"
  R:RCM1 _net12 _net13 R="1M"
  VCVS:ECM1 _net8 _net12 _ref _ref G="31.6228"
  C:CIN1 _net6 _net7 C="1.4e-12"
  R:R2 _net7 _net8 R="1e+06"
  R:R1 _net6 _net8 R="1e+06"
  Idc:IOFF1 _net7 _net6 I="1e-08"
  Idc:IB2 _net7 _ref I="8e-08"
  Idc:IB1 _ref _net6 I="8e-08"
  Vdc:VOFF2 _net7 _netIN_P U="0.00035"
  Vdc:VOFF1 _netIN_N _net6 U="0.00035"
.Def:End
```

Figure 9.16: Qucs netlist for a UA741 AC modular OP AMP model.

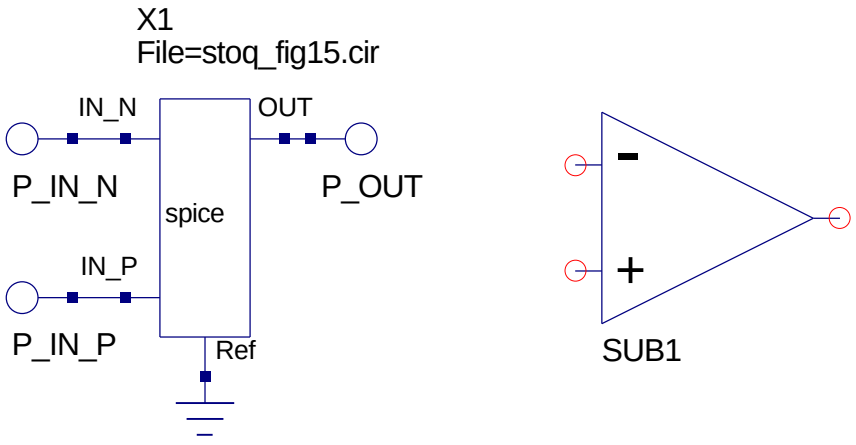


Figure 9.17: Qucs symbol for a UA741 AC modular OP AMP model.

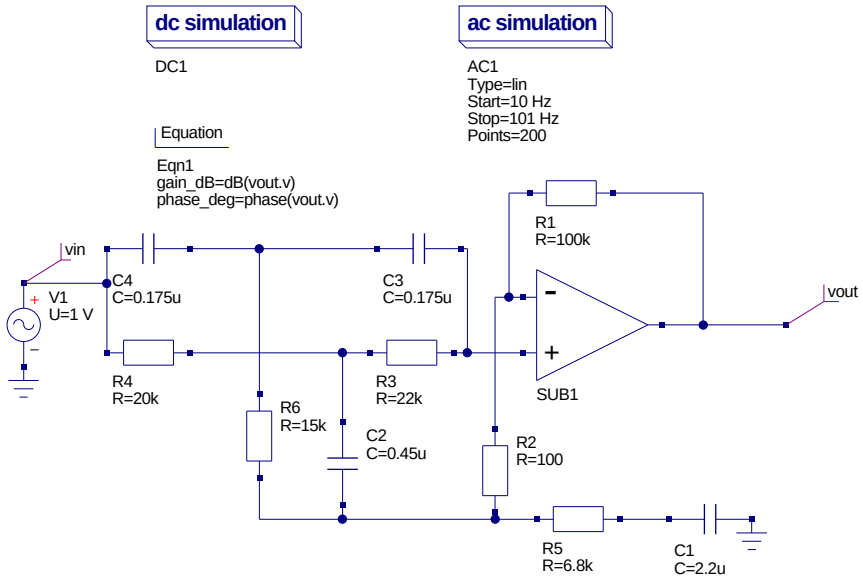


Figure 9.18: A twin-T notch filter circuit.

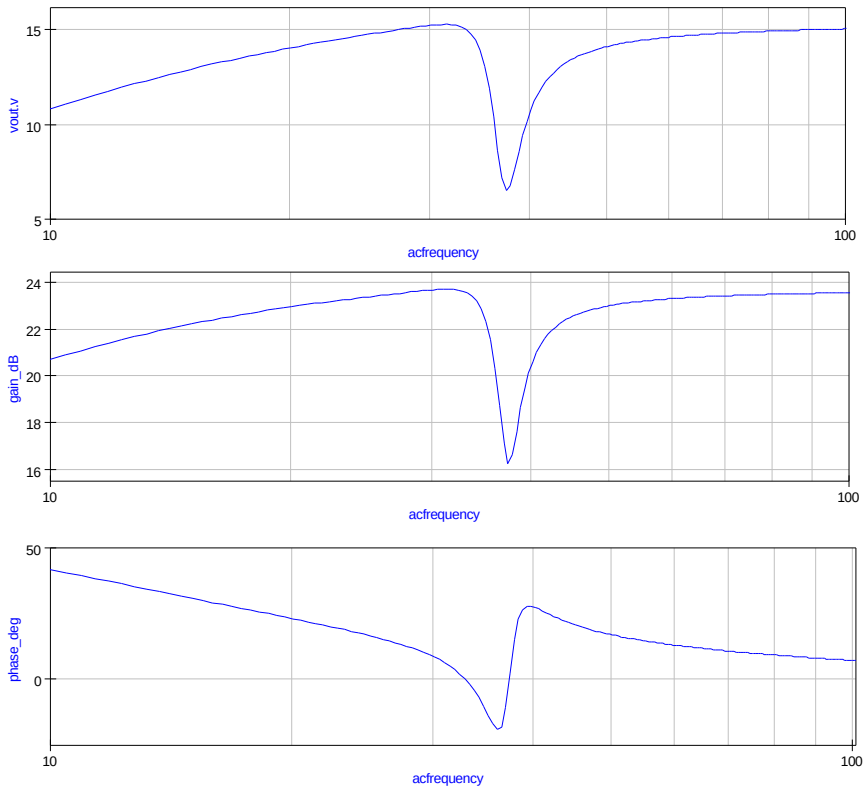


Figure 9.19: Small signal transfer characteristics for a twin-T notch filter circuit.

9.8 Building circuit design equations into netlists

Figure 9.20 illustrates a bandpass filter that has a bandwidth which is small compared to it's center frequency. The circuit is often referred to as the Dalyiannis-Friend filter after its developers. The filter center frequency f_0, voltage gain magnitude H_0, bandwidth B and Q factor are given by the following equations:

- $f_0 = \dfrac{1}{2\pi C\sqrt{(R_1 \| R_2)R_3}}$, where $C = C_1 = C_2$
- $H_0 = \dfrac{R_3}{2R_1}$
- $B = \dfrac{1}{\pi R_3 C}$
- $Q = \dfrac{f_0}{B} = \dfrac{1}{2}\sqrt{\dfrac{R_3}{R_1 \| R_2}}$

When designing a filter for a specific specification, for example say $f_0 = 1kHz$, $B = 200Hz$ and $H_0 = 10$, values for the filter resistor and capacitor values need to be calculated. This can, of course, be done manually. However, this process is often tedious, especially if a number of filters need to be designed each with different specifications. Circuit simulators are by their very nature primarily designed to analyse and simulate the performance of circuits who's component values are known. As such they are tools for analysis rather than design. In practice, of course, engineers employ circuit simulators to check their circuit designs. Qucs is attempting to bridge the gap between design and analysis by using add-on software components for designing circuits with well understood structures and design procedures[23].

In the previous section it was shown that the SPICEPP preprocessor could be used to calculate model component values. By a simple extension of this concept it is also possible to embed design equations into a netlist. Shown in Fig. 9.21 is a SPICEPP netlist for the Dalyiannis-Friend filter. The UA741 OP AMP is modelled with a SPICE subcircuit called `opamp_ac` and has its own set of parameters[24]. The first set of design parameters represent the filter specification and are used in the SPICEPP conversion process to calculate the filter resistor and capacitor component values. Note also the use of inline comments for documenting the netlist code. Figures. 9.22 and 9.23 show a basic filter test circuit and the resulting simulation transfer functions. Hence, not only can the SPICEPP preprocessor be used for setting up device models but it can also aid the design of entire circuit blocks provided design equations are available for a given circuit configuration. By combining SPICEPP with Qucs a very significant design/analysis tool becomes available opening up new possibilities for Qucs users.

[23]The Qucs Tools drop-down menu lists the currently available design functions that have been implemented with release of Qucs you are using.

[24]These are defined within a subcircuit and should have names unique to the subcircuit model being defined.

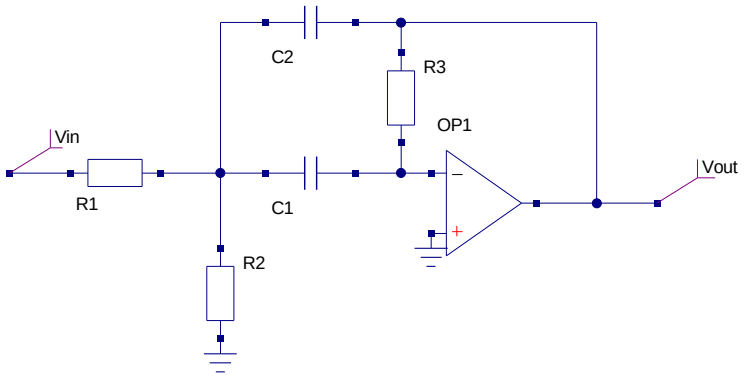


Figure 9.20: The Dalyiannis-Friend bandpass filter circuit.

```
* Delyiannis Friend Bandpass filter design
* Design parameters
.param fc =   2000.0     $ Filter center frequency (Hz)
.param bw  =  200.0      $ Filter bandwidth (Hz)
.param q   =  10.0       $ Filter q factor = f0/bw
.param r3iv = 200k       $ Assumed value for rf3
.param h0 =   10.0       $ Filter f0 gain magnitude
*
* Filter circuit pins: input n1, output n3
*
r3 n3 n4  r3iv
c1 n2 n3 'q/(3.1412*fc*r3iv)'
c2 n2 n4 'q/(3.1412*fc*r3iv)'
r1 n1 n2 'r3iv/(2*h0)'
r2 n2 0  'r3iv/( (4*q*q)-(2*h0) )'
x1 0 n4 n3 opamp_ac

*subcircuit ports: in+   in-   out
.subckt opamp_ac in_p in_n out
*
* ua741 OP AMP parameters
.param voff = 0.7m
.param ib = 80n
.param ioff = 20n
.param rd = 2meg
.param cd = 1.4p
.param cmrrdc = 31622.8
.param fcmz = 200.0
.param aoldc = 199526
.param gbp = 1meg
.param fp2 = 3meg
.param ro = 75.0
* input stage
voff1 in_n 6 'voff/2'
voff2 7 in_p 'voff/2'
ib1   0 6 ib
ib2   7 0 ib
ioff1 7 6 'ioff/2'
r1    6 8 'rd/2'
r2    7 8 'rd/2'
cin1  6 7 cd
* common-mode zero stage
ecm1 12 0 8 0 '1e6/cmrrdc'
rcm1 12 13 1meg
ccm1 12 13 '1/(2 * 3.1412 * 1e6 * fcmz)'
rcm2 13 0 1
* differential and common-mode signal summing stage
gmsum1 0 14 7 6 1
gmsum2 0 14 13 0 1
rsum1  14 0 1
* voltage gain stage 1
gmp1 0 9 14 0 1
rado 9 0 aoldc
cp1 9 0 '1/(2 * 3.1412 * gbp)'
* voltage gain stage 2
gmp2 0 11 9 0 1
rp2  11 0 1
cp2  11 0 '1/(2 * 3.1412 * fp2)'
*
* output stage
eos1 10 0 11 0 1
ros1 10 out ro
.ends
```

Figure 9.21: SPICEPP netlist for the Dalyiannis-Friend filter.

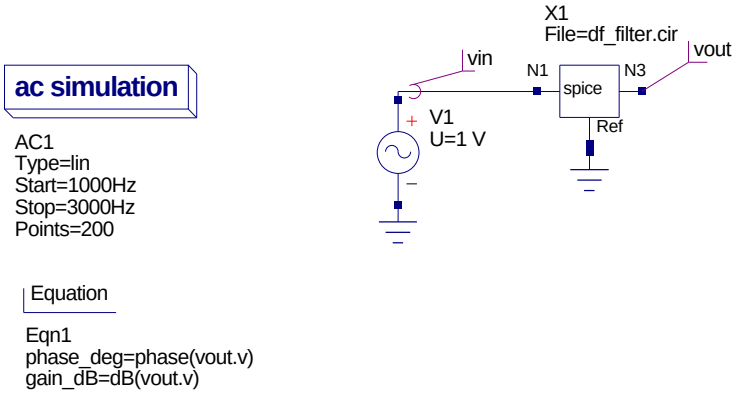


Figure 9.22: The Dalyiannis-Friend bandpass filter test circuit.

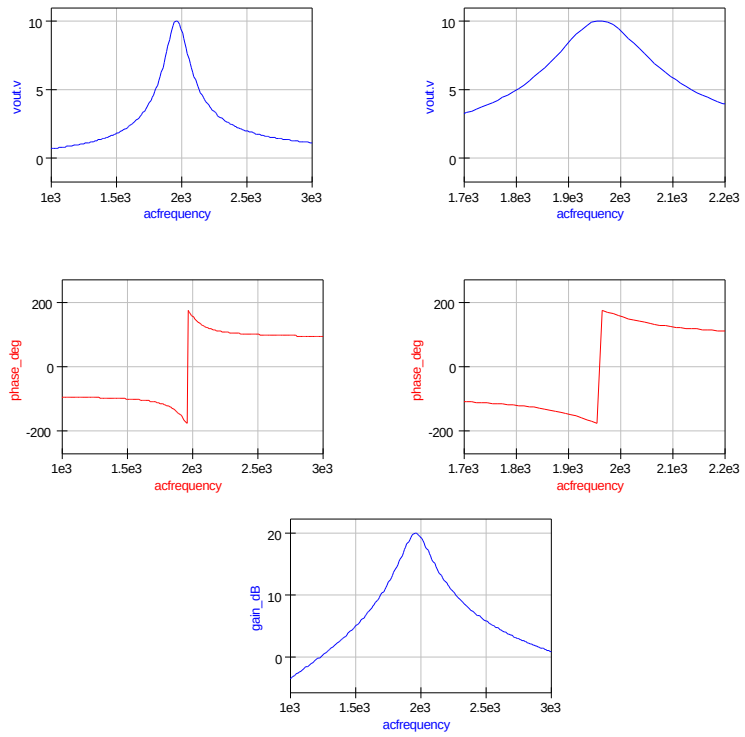


Figure 9.23: Simulated small signal AC transfer functions for the Dalyiannis-Friend bandpass filter.

9.9 Global nodes

In the SPICE 2 and SPICE 3 hardware description languages only the earth node is global. By convention this is given node name 0 and is assumed by the SPICE language passer to be earth whenever it occurs in a circuit netlist. When connecting discreet components with other subcircuit blocks there is often a need for other nodes to be designated global; the classic example being power supply nodes. SPICEPP allows nodes to designated as global. These are effectively connected together to form one net covering both outside and inside subcircuits. The best way to understand the use of global nodes is to consider an example. Figure 9.11 gives the SPICE netlist for the two section CMOS ring counter. Many readers would possibly have noticed that in this netlist both the NAND2 and NOR2 subcircuits include internal voltage sources[25]. This is, of course, not necessary and indeed inefficient from a simulation point of view. A better approach would be to link individual gates with a power supply net. The SPICEPP netlist given in Fig. 9.24 illustrates how the .global command can be used to define a global power supply node. After passing this code through SPICEPP the SPICE netlist printed in Fig. 9.25 results. Simulation with Qucs gives the same waveforms displayed in Fig. 9.13.

[25]The DC voltage supply for each logic block is generated by a pulse source. This has the effect of simulating the rising edge of the power supply switch on transient and aids DC convergence.

```
* Two stage CMOS ring counter circuit.
*
* External nodes: input 1, output 4, +ve supply  nvcc
*
*global node
*
.global nvcc
*
x1 1 5 6  nand2
x2 1 6 7  nand2
x3 3 6 2  nand2
x4 2 7 3  nand2
x5 1 2 8  nor2
x6 1 8 9  nor2
x7 5 8 4  nor2
x8 4 9 5  nor2
*
.model modp pmos( vto=-1 kp=10u
+                          cgdo=0.2n cgso=0.2n cgbo=2n)
.model modn nmos(vto=1 kp=10u
+                          cgdo=0.2n cgso=0.2n cgbo=2n)
*
.subckt nand2 1 2 3
m1 3 1 nvcc nvcc modp w=40u l=5u
m2 3 2 nvcc nvcc modp w=40u l=5u
m3 5 1 0 0 modn w=20u l=5u
m4 3 2 5 5 modn w=20u l=5u
c1 1 0 10p
c2 2 0 10p
*vcc 4 0 pulse ( 0 5 0 1ns 1ns 1 2)
.ends
*
.subckt nor2 1 2 3
m1 4 1 nvcc nvcc modp w=40u l=5u
m2 3 2 4 4 modp w=40u l=5u
m3 3 2 0 0 modn w=20u l=5u
m4 3 1 0 0 modn w=20u l=5u
c1 1 0 10p
c2 2 0 10p
*vcc 7 0 pulse ( 0 5 0 1ns 1ns 1 2)
.ends
```

Figure 9.24: SPICEPP netlist for a two section CMOS ring counter with global power supply net node nvcc.

```
* Two stage CMOS ring counter circuit.
x1 1 5 6 nvcc nand2
x2 1 6 7 nvcc nand2
x3 3 6 2 nvcc nand2
x4 2 7 3 nvcc nand2
x5 1 2 8 nvcc nor2
x6 1 8 9 nvcc nor2
x7 5 8 4 nvcc nor2
x8 4 9 5 nvcc nor2
.model modp pmos vto=-1 kp=10u cgdo=0.2n cgso=0.2n cgbo=2n
.model modn nmos vto=1 kp=10u cgdo=0.2n cgso=0.2n cgbo=2n
.subckt nand2 1 2 3 nvcc
m1 3 1 nvcc nvcc modp w=40u l=5u
m2 3 2 nvcc nvcc modp w=40u l=5u
m3 5 1 0 0 modn w=20u l=5u
m4 3 2 5 5 modn w=20u l=5u
c1 1 0 10p
c2 2 0 10p
.ends
.subckt nor2 1 2 3 nvcc
m1 4 1 nvcc nvcc modp w=40u l=5u
m2 3 2 4 4 modp w=40u l=5u
m3 3 2 0 0 modn w=20u l=5u
m4 3 1 0 0 modn w=20u l=5u
c1 1 0 10p
c2 2 0 10p
.ends
```

Figure 9.25: SPICE netlist for a two section CMOS ring counter with global power supply net node nvcc.

9.10 End Note

This tutorial note describes how SPICE netlists can be simulated using Qucs. The text is much more than a basic outline of the processes needed to link SPICE circuit files to Qucs. While writing this note an attempt has been made to stress the fact that topics like SPICE/Qucs netlist compatibility and conversion are important to the future development of Qucs. So an interesting, and thought provoking question, is how does Qucs develop next in relation to SPICE and indeed how best is it to make sure that Qucs users can get the most from all the published SPICE information and device models? After all there is no point in reinventing the wheel! Complete compatibility with SPICE will not be possible until all the basic SPICE 2 and SPICE 3 primitive components are added to Qucs. This will take time but is happening as the Qucs team develops the package[26]. Adding equations to component calculations is a very much a current active topic in Qucs development. Recently, Michael Magraf has added parameter passing to the Qucs GUI. Stefan Jahn will add the necessary simulator routines for handling equations and parameter passing when time allows. In the long term not only will it be possible to determine component values using calculations at the simulation initialisation phase but it will also be possible to allow such components to be dependent on simulation voltage and current variables. Qucs will

[26]Michael Magraf has recently added a four terminal transmission line to Qucs. Future testing will confirm if this is similar to the SPICE T component.

then be able to simulate circuits containing nonlinear voltage and current sources like the SPICE 3 B component. These notes are very much a report on some of the work on Qucs device modelling I have been doing in recent months. Again if there is enough interest in this area of Qucs development I will upgrade them in the future. My thanks to Stefan Jahn for all his encouragement while I have been developing the material reported in this tutorial note.

10 Biasing a BJT Transistor

10.1 Graphical methods

You can bias a bipolar junction transistor in several ways. Determining the best method for your application is easy with a graphical technique.
Biasing an active device, such as a bipolar junction transistor (BJT), requires that you set the dc voltages and currents of the device. To optimize the desired result, you need various bias values. For instance, the input de-vice for a low-noise amplifier may have its best noise performance at 50 μA of collector current and a maximum of 5V of collector-to-emitter voltage, whereas later amplifier stages may require 20-mA collector current and 18V collector-to-emitter voltage to generate the necessary ac voltage at the output. When you determine the desired bias conditions, you also need to make sure they are repeatable–within certain limits–to ensure consistent performance.

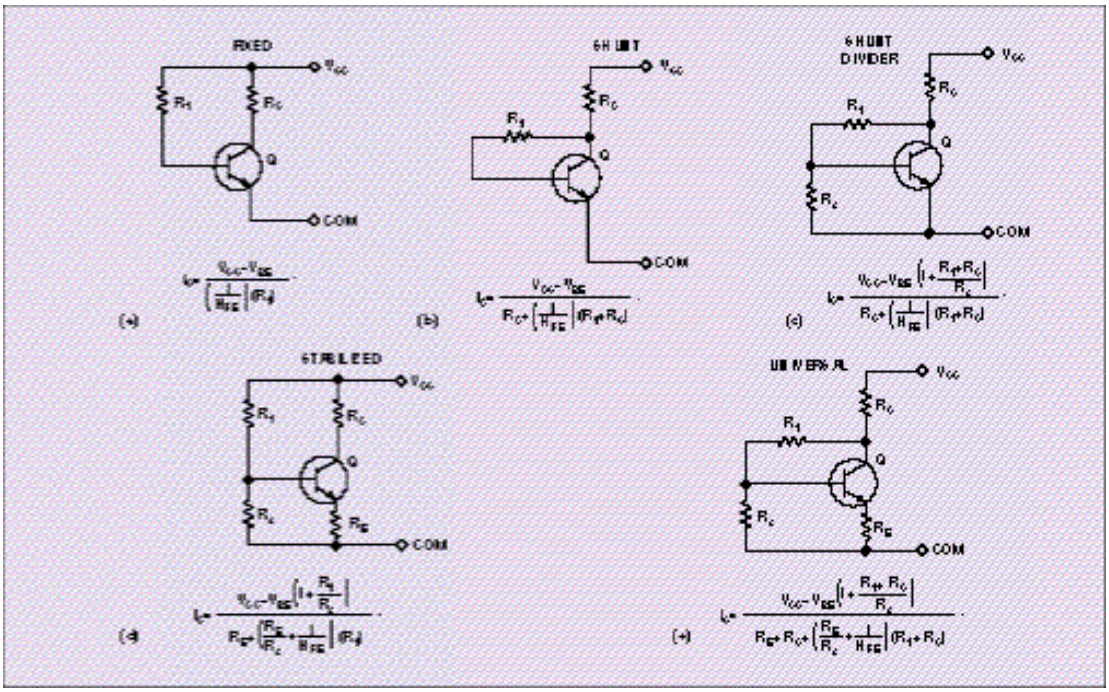


Figure 10.1: Different feed back technics

Biasing-technique analysis for BJTs generally progresses in complexity from the fixed-bias

method (see fig 10.1, to the shunt circuit, to the stabilized circuit, . Studies do not usually cover the shunt-divider and universal circuit. However, questions still arise about the bias stability of the shunt bias circuit. It is usable in some noncritical applications, but how inferior is it to the stabilized circuit? Designers are generally taught that the stabilized circuit is the one to use for repeatable biasing.

One way to analyze the stability of the various biasing methods is to use stability factors, which characterize the change in collector current due to changes in the transistor's HFE (current gain), ICBO (collector-to-base leakage current), and VBE . Although these factors are useful, comparing bias circuits and bias-resistor values requires tedious calculations. A visual presentation that compares the stability of the various circuits is more useful.
Looking at the equation for IC in Figure 1b, note that much of the change in IC is due to the differing voltages developed across R1 because of the range of HFE. This difference leads to a question: If some of the current through R1 is fixed, would the result be less voltage change across R1 and hence, less change in IC? This thinking leads to the shunt-divider circuit (Figure 1c). Because VBE changes little, R2 supplies a relatively fixed component of the current through R1, making R1 a smaller value than it would be without R2. The equation for the shunt divider shows that a smaller value of R1 in the denominator causes less change in IC due to changes in HFE. However, along with RC and R2, R1 shows up in the numerator as a multiplying factor for VBE.

You can next look at how strongly each of these factors influences IC. Because you can derive all the circuits in Figure 1 from the universal circuit (Figure 1e) by making the appropriate resistors either infinite (open circuits) or zero (short circuits), the same universality is possible for the equations. Considering the circuit equations and a range of parameters and bias-resistor values, you can produce graphs in which the Y axis represents the change in IC.
To make valid comparisons of the circuits, you need a common parameter related to the biasing for the X axis. The ratio of the collector current to the bias current in R1 works. This ratio is common to the circuits and reflects how stiff the biasing is. To show realistic conditions, the data also includes temperature effects on VBE and HFE for a temperature range of 25 to 75°C and a 3-to-1 spread in HFE.
For comparison purposes, all the circuits use a 10V supply for VCC at a nominal collector current of 1 mA, with HFE of 100 and VBE of 0.60V at 25°C. Calculating resistors for 5V VCE and selecting RE to develop 1V at the emitter produces the results for the graphical technique. The model for temperature effects of the device is VBE=0.60?0.002?(T(actual) 25°C), representing the standard 2-mV/°C coefficient for diodes. Calculations from the data sheet of the 2N2222A transistor produce an average temperature coefficient for HFE of about 0.58% /°C, which you can represent by

$$HFE_{Temp} = HFE_{Max} \times [1 + (T(actual)?25°C)0.0058] \tag{10.1}$$

Calculating IC for a minimum $HFE = 50$ at 25°C and for the maximum $HFE = 150$ at 75°C yields an HFE_{Temp} of 194 and VBE of 0.50V.

This analysis ignores the effects of ICBO. For the nominal collector current of 1 mA and a maximum temperature of 75°C, the contribution of ICBO to IC is a few percent, at most, for the fixed-bias and shunt-bias circuits in Figures 1a and 1b and less for the bias circuits of Figures 1c, 1d, and 1e.

10.1.1 Graphical approach shows trade-offs

The results of this analysis appear as a simple visual comparison of the current stability of the various types of bias circuits (Figure 10.2). Using this figure, you can select the type of bias circuit and the bias ratios for the necessary stability.

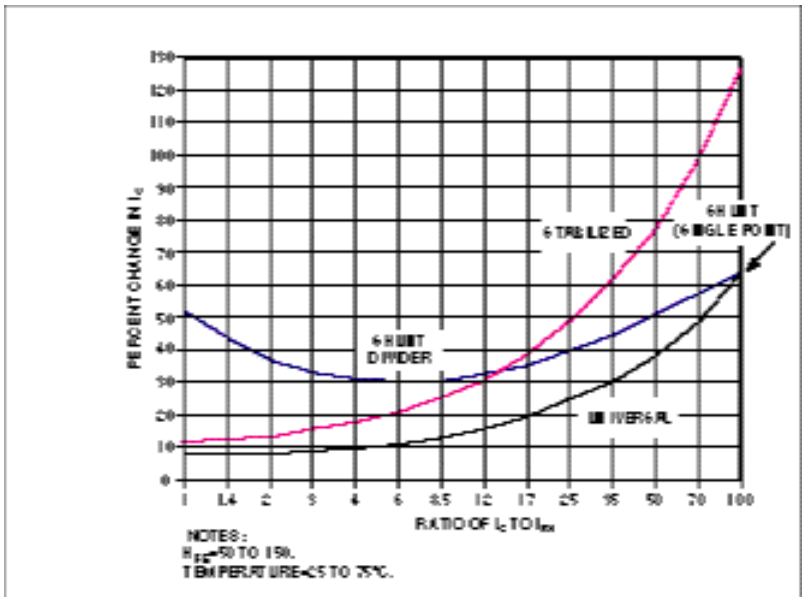


Figure 10.2: You can compare the performance of the BJT bias circuit by graphing the change in collector current vs the ratio of the collector current to the current in $R1$.

The horizontal axis is the ratio of the collector current, IC, to the current in resistor R1. This bias ratio applies to all the circuits and indicates how much current is in the base-biasing network compared with the collector current. Thus, a ratio of 1 indicates a stiff bias circuit, with as much current in R1 of the bias network as in the collector, whereas a ratio of 50 indicates that the collector current is 50 times the current in R1 of the bias network. Because some of the results are unexpected, they give renewed consideration to some of the bias circuits previously ignored.

The universal-bias method is obviously the best of the group. The price you pay for its dc stability is the reduction in ac input resistance due to the negative feedback on R1, a sort of Miller effect on resistors. R1 reduces by a factor of the voltage gain plus 1. This feedback may improve distortion and bandwidth as well as reduce the output impedance

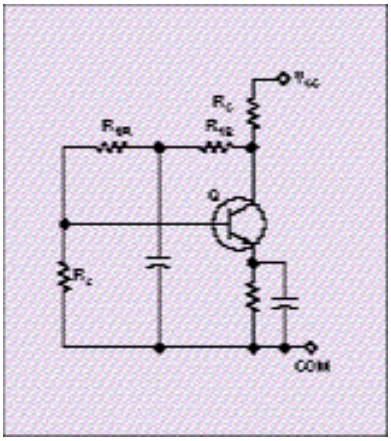


Figure 10.3: To eliminate the ac effects of feedback, split $R1$, and bypass the center to ground.

at the collector. If you don't want these ac effects of feedback, you can eliminate them by splitting R1 into two parts and bypassing the center to ground (Figure 10.3). You can improve performance of this circuit at any bias ratio by increasing the voltage drop across RE, increasing the voltage drop across the collector resistor, or both.

The stabilized circuit has good stability to bias ratios as high as about 12. Above this ratio, its stability rapidly decreases. The stabilized circuit relies on the voltage changes fed back by the emitter current through RE, compared with the voltage, VB, at the base. When the bias ratio becomes less stiff, changes in base current flowing through R1 due to changes in HFE cause significant variations in VB. These variations result in changes in IE and IC. As with the universal circuit, you can improve performance of the stabilized circuit at any bias ratio by increasing the voltage drop across RE. Keep in mind that these results are for a nominal HFE range of 50 to 150 plus temperature effects. Lower minimum values of HFE require stiffer bias ratios for the same performance.

The superior performance of the shunt-divider circuit at bias ratios greater than 12, compared with that of the stabilized circuit, is a surprise. When the shunt-divider circuit's bias is stiff, VC is strongly influenced by the ratio of R1 to R2 times VBE. As VBE changes because of temperature, VC and, thus, IC, change approximately as the ratio of R1 to R2 times VBE changes.

Because IC plays the major role in determining VC, IC experiences wide variations for these stiff biasing ratios. As the ratio becomes less stiff, the changes in VBE with temperature, multiplied by the voltage-divider action, become less dominant, and performance improves until, at the ratio of about 12, the shunt divider's stability starts to surpass that of the stabilized circuit. You can account for this performance by the negative feedback from the collector resistor through R1. Because the collector resistor is usually much larger than

the emitter resistor of the stabilized circuit, the stability of the universal circuit holds up better for less stiff bias ratios.

Because the shunt-divider circuit is more stable than the shunt circuit, consider the divider circuit for applications that need less stability than the stabilized or universal circuits offer. Because it saves the cost of the emitter-bypass capacitor necessary in the universal and stabilized circuits, the shunt divider can be more cost-effective. Negative feedback through R1 in the shunt-divider circuit reduces the input resistance and may improve distortion and bandwidth, as well as reduce the output impedance in the same manner as in the universal circuit. Again, you can negate these effects with a bypass capacitor in the center of R1. This bypass capacitor is typically much smaller than the emitter-bypass capacitor for the stabilized circuit.

Because the bias current for the shunt-bias circuit consists of only the base current, it has only one ratio of IC to IR1, namely HFE, and is plotted as a single point. As the bias ratio for the universal and shunt-divider circuits increases, the value of R2 increases until it becomes infinite at an HFE of 100. Under these conditions, the circuits' bias ratios converge with the shunt-circuit ratio.

Figure 10.2 leads you to several general conclusions. The universal circuit has the best stability over the widest range of bias ratios. The stabilized circuit has good stability for stiff bias ratios, but you should take care if biasing ratios exceed 12. And, finally, the shunt-divider circuit is a significant improvement over the shunt circuit and is better than the stabilized circuit for large bias ratios.

10.2 Simulation technics

The previous section deals with a graphical method, but a more common method can be to use the simulators to determine all the possible variation for a given schematic (include h_{FE}, Temperature, Voltage regulation, and so on ...) ; so the problem is more waht kind of feedback I can use or not. Sorry but there is no striaght ansyert since this could a cost issu e for example, or a performance issue[1].

Anyway we need to evaluate the different biasing technics using the simulation tool. One analysis will be done in the PA design chapter.

[1]This point is obviously not understood in the same way when discussing with marketing or development or research teams, who knows why ?

11 BJT Modeling and Verification

warning

This chapter will describe an RF design issue using QUCS. The author assume that the basic manipulation of qucs is known. You will find herein mainly a MacOsX description that is close to a linux or unices architecture.

11.1 choice of transistor

The choice has been made to choose among the Philips RF wideband transistor library. These components are easy to find, with resonnable prices.
This list could be found at `http://www.semiconductors.philips.com/`.
A resume of these transistors can be found in the figure 11.1
I will not discuss herein, the reason [1] why of the final choice, but the $BFG425w$ is the candidate. It offers high gain, with low figure noise (if LNA consideration) high transistion frequency (25 GHz), its emitter is thermal lead, low feedback capacitance. This device could be used in RF front end, analog or digital cellular, radar detectors, pagers, SATV, oscillators. It is in a SOT343R package suitable for small integration.
The maximum acheivable gain is 20 dB with 25 mA, $V_{ce} = 2$ V at 2 GHz and $25°C$. The third order intercept point in these conditions is typically $22dBm$.
These parameter should be compatible with our need. Here are the spice parameter of the device.

```
.SUBCKT BFG425W 1 2 3
L1   2 5 1.1E-09
L2   1 4 1.1E-09
L3   3 6 0.25E-09
Ccb  4 5 2.0E-15
Cbe  5 6 80.0E-15
Cce  4 6 80.0E-15
Cbpb 5 7 1.45E-13
Cbpc 4 8 1.45E-13
Rsb1 6 7 25
Rsb2 6 8 19
```

[1] regarding current, F_t , V_{ce} , power dissipation, etc ...

Selection results for RF wideband transistors

Go to the interactive version of this Selection Guide.
Download in MS-Excel sheet (right-click and choose Save Target As...)

80 RESULTS SHOWN.

Type number	PACKAGE	Category	POLARITY	P_{TOT} (mW)	I_C (mA)	f_T (GHz)	VCEO max (V)	@f1	@f (MHz)	G_{UM} @f1 (dB)	Frequency (MHz)	NOISE FIGURE MAX (dB)	G_{UM} @f2 (dB)	@f2	V_{CE} (V)	@I_C (mA)	ITO (dBm)	Socket	V_O ² (mV)	P_L (W)	V_o	Gain @900 Mhz (dB)	G_{UM} (dB)	Gain @1.9 Ghz (dB)	System Freq (Mhz)
BFG10	SOT143B	Transistor wideband NPN up to 3.5 GHz	NPN	250.0	250.0		8.0	1900.0	1900	7.0												7		
BFG10/X											900														
BFG10W/X	SOT343N																								
BFG135	SOT223	Transistor wideband NPN up to 8 GHz	NPN	1000.0	150.0	7.0	15.0	500.0	500	16.0			12	800	10	100			850	850		16			
BFG198					100.0	8.0	10.0			18.0			15		8	70			700	700		18			
BFG21W	SOT343R	Transistor wideband NPN up to 25 GHz									1900														
BFG25A/X	SOT143B	Transistor wideband NPN up to 6 GHz	NPN	32.0	6.5	5.0	5.0	1000.0	1000	18.0	1.8	1.8@f1										18			
BFG25AW/X	SOT343N			500.0								2.0@f1	8	2000											
BFG31	SOT223	Transistor wideband PNP up to 6 GHz	PNP	1000.0	100.0		15.0	500.0	500	16.0	2		12	800	10	70			550	550		16			
BFG35		Transistor wideband NPN up to 6 GHz			150.0	4.0	18.0			15.0			11			100			750	750		15			
BFG403W				16.0	3.6	17.0				1.6		1.6@f2 1.0@f1 1.6@1.9GHz 1.0@900MHz		1		1	6					20.0		22.0	900
BFG410W	SOT343R	Transistor wideband NPN up to 25 GHz		54.0	12.0	22.0	4.5		900		1.2	1.2@f2 0.9@f1 0.9@1.9GHz 1.2@900MHz				10	15	LNA			5	21.0		29.0	900 & 1900
BFG425W				135.0	30.0	25.0		900.0				1.2@f2 0.8@f1 0.8@1.9GHz 1.2@900MHz		2000	2	25	22			12		20.0		28.0	
BFG480W				360.0	250.0	21.0						1.8@f2 1.2@f1				80	28							16.0	
BFG505	SOT143B		NPN	150.0						20.0			13									20			
BFG505/X					18.0											5	10			4					
BFG505W/X	SOT343N			500.0							1.9	1.9@f2 1.6@f1	12					LNA, Mixer, Buffer & VCO					19		
BFG520										19.0			13									17.0		10.0	
BFG520/X	SOT143B			300.0																275					
BFG520/XR															6										
BFG520W	SOT343N				70.0	9.0	15.0		900			1.8@f2 1.6@f1 1.9@1.9GHz 1.2@900MHz				20	26		275	17					
BFG520W/X		Transistor wideband NPN up to 10 GHz		500.0				900.0		17.0	1.85	1.8@f2 1.6@f1	11	2000				LNA, Mixer, Buffer & VCO		275		17.0		10.0	
BFG540	SOT143B											2.1@f2 1.9@f1								500					
BFG540/X					120.0					18.0	2.1			8		40	34		500	21					
BFG540/XR	SOT143R																								
BFG540W											900														
BFG540W/X	SOT343N			500.0				900.0		16.0		2.1@f2 1.9@f1	10	2000						500					

Figure 11.1: transistor table from philips semiconductor

```
Q1 4 5 6 6 NPN

.MODEL NPN NPN
+ IS = 4.717E-17  + BF = 145    + NF = 0.9934
+ VAF = 31.12  + IKF = 0.304 + ISE = 3.002E-13
+ NE = 3  + BR = 11.37  + NR = 0.985
+ VAR = 1.874  + IKR = 0.121  + ISC = 4.848E-16
+ NC = 1.546  + RB = 14.41 + IRB = 0
+ RBM = 6.175  + RE = 0.1779  + RC = 1.780
+ CJE = 3.109E-13  + VJE = 0.9  + MJE = 0.3456
+ CJC = 1.377E-13 + VJC = 0.5569 + MJC = 0.2079
+ CJS = 6.675E-13 + VJS = 0.4183 + MJS = 0.2391
+ XCJC = 0.5  + TR = 0.0  + TF = 4.122E-12
+ XTF = 68.2  + VTF = 2.004  + ITF = 1.525
+ PTF = 0  + FC = 0.5501  + EG = 1.11
+ XTI = 3  + XTB = 1.5
.ENDS
```

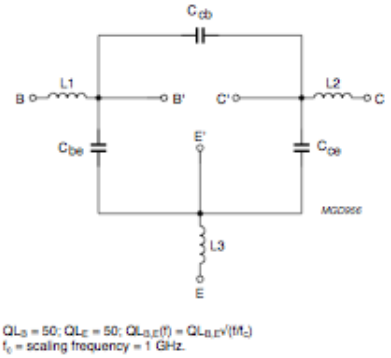
Since the model used in SPICE and in QUCS rely on a gummel-poon modelisation, and since the level of modelisation is the same, some quite direct conversion could be used to create the library for QUCS.
To use directly this file, you will need to store the file in an other directory from the project one (a small bug taken into account). Then it should work but some there are still some issues on the parameters itselves, This is the reason why we will proceed in an other way. The data sheet could be found on the philips web site.

SPICE parameters for the BFG425W die

SEQUENCE No.	PARAMETER	VALUE	UNIT
1	IS	47.17	aA
2	BF	145.0	–
3	NF	0.993	–
4	VAF	31.12	V
5	IKF	304.0	mA
6	ISE	300.2	fA
7	NE	3.000	–
8	BR	11.37	–
9	NR	0.985	–
10	VAR	1.874	V
11	IKR	0.121	A
12	ISC	484.8	aA
13	NC	1.546	–
14	RB	14.41	Ω
15	IRB	0.000	A
16	RBM	6.175	Ω
17	RE	177.9	mΩ
18	RC	1.780	Ω
19 (1)	XTB	1.500	–
20 (1)	EG	1.110	eV
21 (1)	XTI	3.000	–
22	CJE	310.9	fF
23	VJE	900.0	mV
24	MJE	0.346	–
25	TF	4.122	ps
26	XTF	68.20	–
27	VTF	2.004	V
28	ITF	1.525	A
29	PTF	0.000	deg
30	CJC	137.7	fF
31	VJC	556.9	mV
32	MJC	0.207	–
33	XCJC	0.500	–
34 (1)	TR	0.000	ns
35 (1)	CJS	667.5	fF
36 (1)	VJS	418.3	mV
37 (1)	MJS	0.239	–
38	FC	0.550	–
39 (2)(3)	C_{bp}	145	fF
40 (2)	R_{sb1}	25	Ω
41 (3)	R_{sb2}	19	Ω

Notes

1. These parameters have not been extracted, the default values are shown.
2. Bonding pad capacity C_{bp} in series with substrate resistance R_{sb1} between B′ and E′.
3. Bonding pad capacity C_{bp} in series with substrate resistance R_{sb2} between C′ and E′.



$QL_B = 50$; $QL_C = 50$; $QL_{B,E}(f) = QL_{B,E}\sqrt{(f/f_c)}$
f_c = scaling frequency = 1 GHz.

Fig.14 Package equivalent circuit SOT343R2.

List of components (see Fig.14)

DESIGNATION	VALUE	UNIT
C_{be}	80	fF
C_{cb}	2	fF
C_{ce}	80	fF
L1	1.1	nH
L2	1.1	nH
L3 (note 1)	0.25	nH

Note

1. External emitter inductance to be added separately due to the influence of the printed-circuit board.

Figure 11.2: spice parameter extract from philips data sheet

11.2 library creation

Remember that when creating a device, it is almost always mandatory to read of have a look at on how the model is done is the technical documentation. It is very to understand the limitation, and how we can correct some data if needed. The mian pity is that a lot of commercial software are quite obscure on the real model they use and their limitation ; QUCS is quite exceptionnal on this point this the complete modeling is explain theoretically in a special technical paper.

In order to conduct these test, we need to create a model of our component. To perform this you should create the file that contain all the libraries, this file is stored under

`/usr/local/share/qucs/library/philips_RF_widebande_npn.lib`

You can edit this file with `vi`. You need to add the following line :

```
<Qucs Library 0.0.7 "philips RF wideBand">

<Component BFG425W>
  <Description>
    RF wideband NPN 25GHz
    2V, 25mA, 20dB , 2000MHz
    Manufacturer: Philips Inc.
    NPN complement: BFG425W
    --------------------------
    based on spice parameter from philips
    --------------------------
    sept 2005  thierry
  </Description>
  <Model>
<_BJT T_BFG425W_ 1 480 280 8 -26 0 0 "npn" 1 "47.17e-10"
1 "1" 1 "1" 1 "0.304" 1 "0.121" 1 "31.12" 1 "1.874" 0
"300.2e-15" 1 "3" 1 "484.8e-10" 1 "1.546" 1 "145" 1 "11.37"
1 "6.175" 1 "0" 1 "1.78" 1 "0177.9e-3" 1 "014.41" 1 "310.9e-15"
1 "0.900" 1 "0.346" 1 "137.7e-15" 1 "0.5569" 1 "0.207" 1 "0.500"
1 "667.5e-15" 1 "0.4183" 1 "0.239" 1 "0.550" 1 "4.122e-12" 1
"68.2" 1 "2.004" 1 "1.525" 1 "0.0" 1 "26.85" 1 "0.0" 0 "1.0" 0
"1.0" 0 "0.0" 0 "1.0" 0 "1.0" 0 "0.0" 0>
  </Model>
</Component>
```

You can replace the 1 by 0, this will remove the visible checkbox, the fact to place a 1 first enable the user to change and or view the parameters that are being used.
A trick to provide all the required syntax is to fill a NPN into the schematics, perform a copy on the device, you should then have the model in the clipboard, just paste into to file

and add the description and the markup language boundaries. The syntaxe is explained in the help at the topic *description of the qucs file formats*.
Then the device is visible in the `Component Library Tool` as mentionned in figure 11.3.

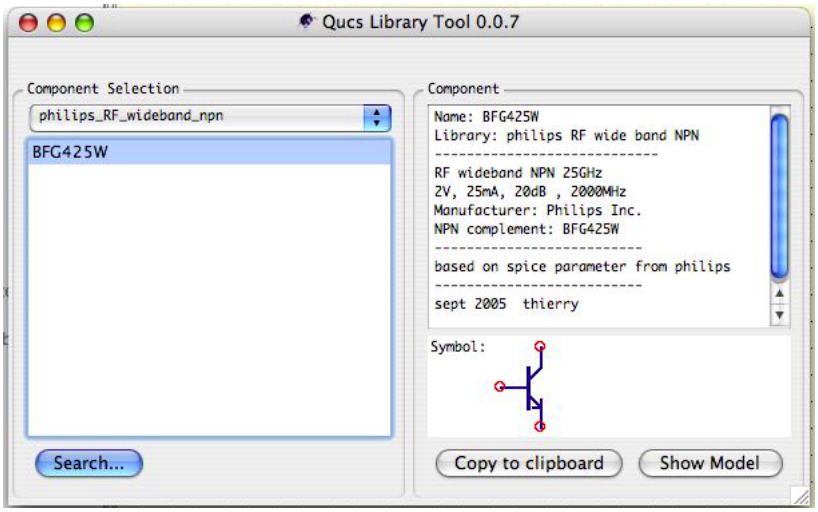


Figure 11.3: QUCS Component Library showing the new component

By doing this you haved the possibility to reuse the device as much as you want, and you can debug devices in a more easy way.
Warning : in this section we have only describe the die of the device, for the parasitic from the package, we will be obliged to describe this circuit, but later on.

11.3 device library verification

The first step, before using the device in a application, is to verify the model you use. Especially since this model has been created by the user. In order to proceed, you need to rely on exact data : that is to say the official datasheet.
it this step, you will need to create a project especially for the device verification. It is good to keep a trace of the device verification, since you could have different use of this device, so it is good to be able to redo some simulation around the model itself.
The created project should look that the figure 11.4.

```
project name :  model_verif_bfg425w
project location : $HOME/.qucs/
```

For the validation we will need to use a specific bias of the device : I_c should be $25mA$, therefore I_b should be $300\mu A$

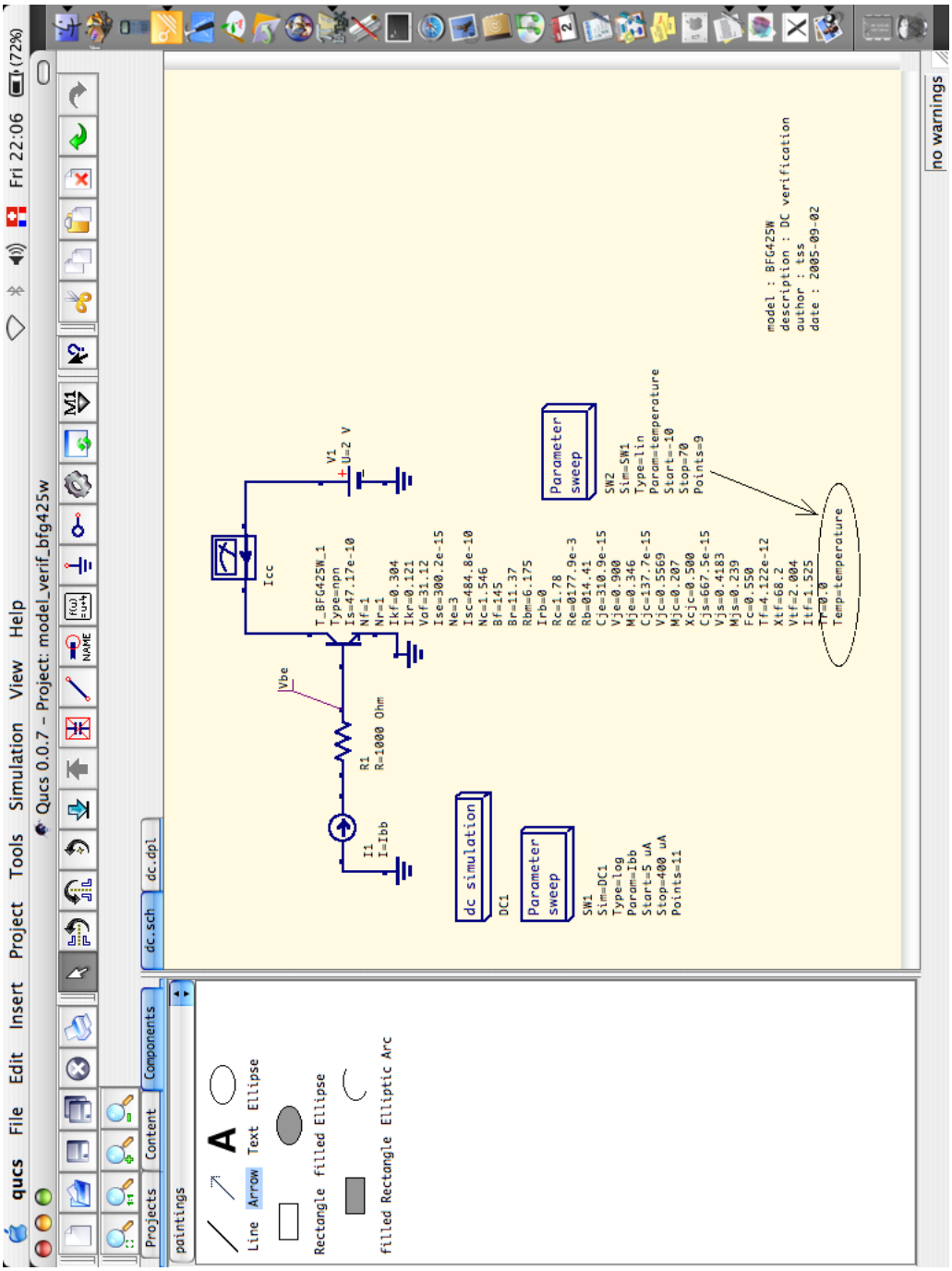


Figure 11.4: QUCS project for model verification

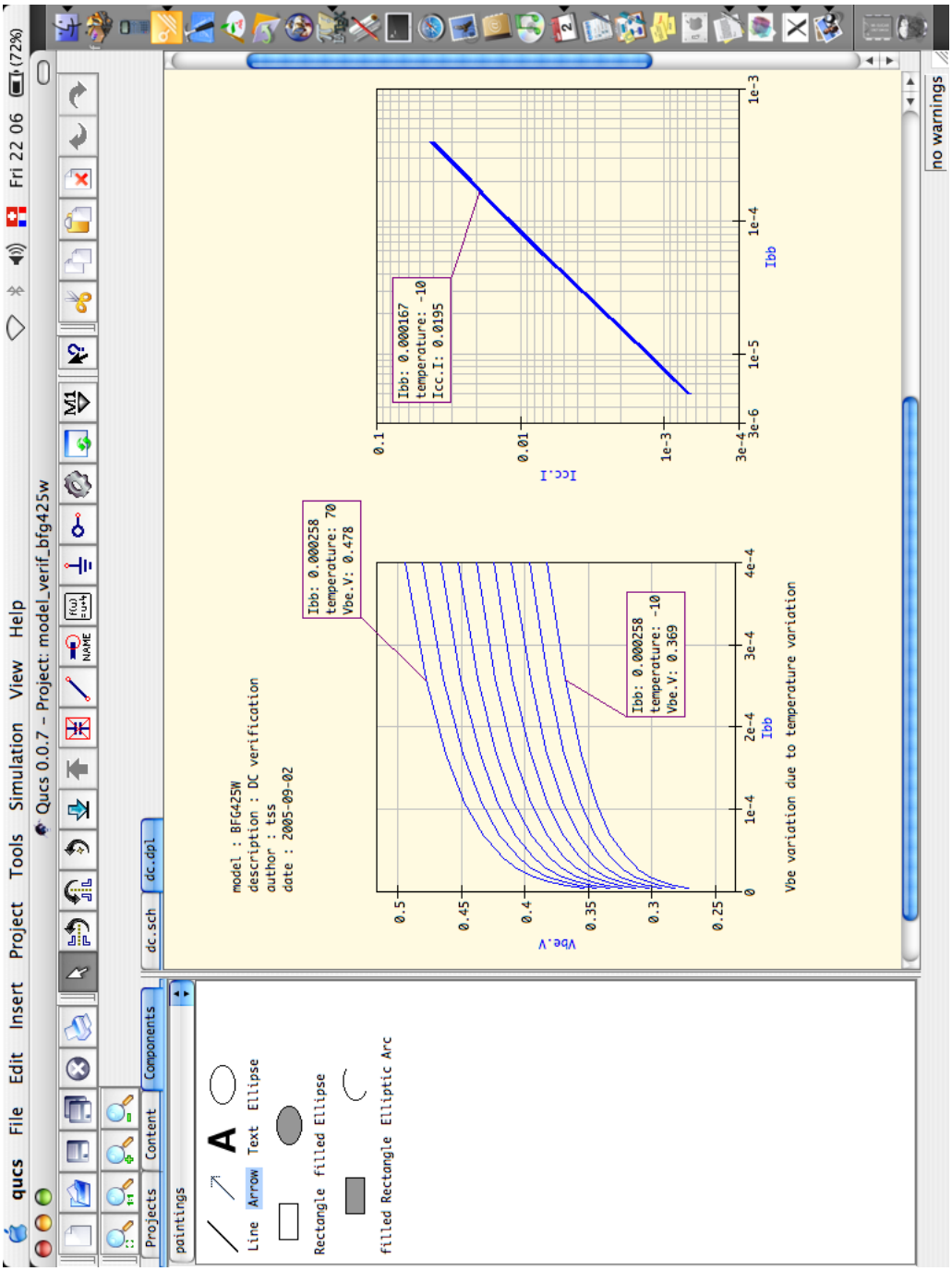


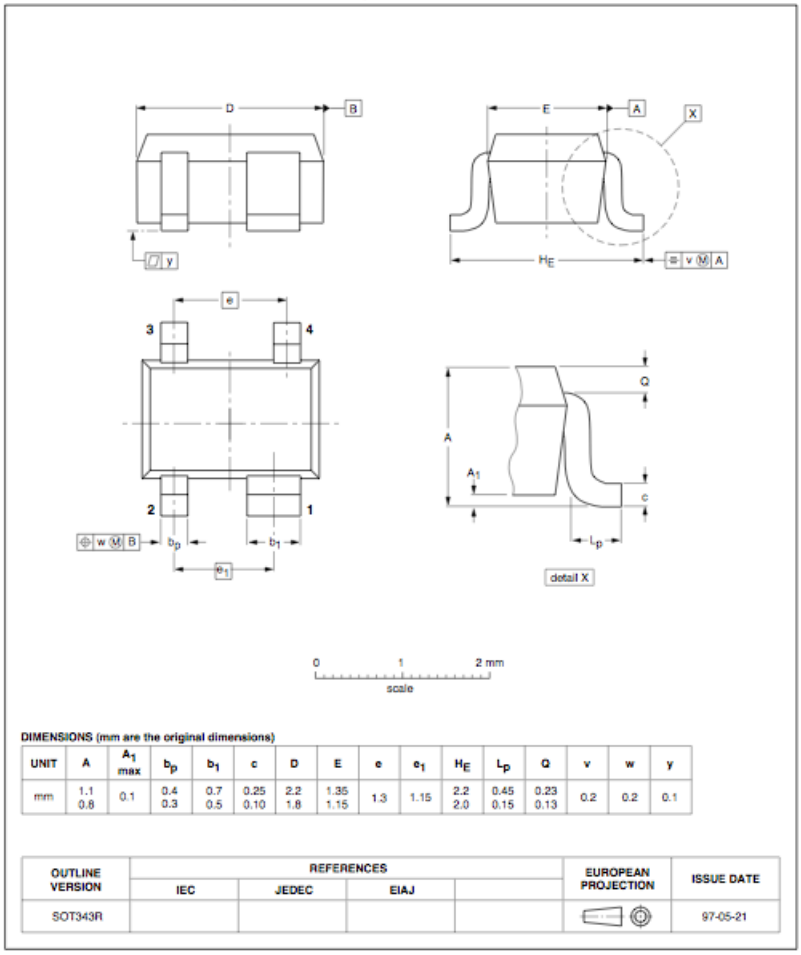
Figure 11.5: DC validation and temperature

11.4 parasitic description of the package

In order to simulate properly the device, you need to used the correct package, that is to say the $SOT343R$ in our case, as mentionned on the philips web site (see fig. 11.6).
Eventhough the device has two emitter, the model used has only one emitter. The parasitic of this model are shoyn in the spice netlist described in the choice of the transistor and reproduced in a schematic (see fig. 11.8). These parameter are always critical to extract, either you have the knowledge to do it or then you should rely on the piece of information given by the device manucfacturer. It is also very difficult to figure out what have to be changed in such description of the device. Some fitting have been performed using 3D electromagnetic software in the time domain based on MOM methods to verify these parameters.
PhilipsÕ fifth generation double poly silicon wideband technology uses a steep emitter doped profile resulting in transition frequencies over 20 GHz, and with poly base contacts a low base resistance is obtained. Via the buried layer, the collector contact is brought out at the top of the die. The substrate is connected directly to the emitter package lead, resulting in improved thermal performance (see fig 11.7).
From this schematics you can edit the symbol that could be used in the next simulation file. To proceed type $F3$ or edit circuit symbol from the file menu. Simply drw a npn transistor and come back to the schematic by re-pressing $F3$.

Philips Semiconductors **Package outline**

Plastic surface mounted package; reverse pinning; 4 leads **SOT343R**



DIMENSIONS (mm are the original dimensions)

UNIT	A	A_1 max	b_p	b_1	c	D	E	e	e_1	H_E	L_p	Q	v	w	y
mm	1.1 0.8	0.1	0.4 0.3	0.7 0.5	0.25 0.10	2.2 1.8	1.35 1.15	1.3	1.15	2.2 2.0	0.45 0.15	0.23 0.13	0.2	0.2	0.1

OUTLINE VERSION	REFERENCES				EUROPEAN PROJECTION	ISSUE DATE
	IEC	JEDEC	EIAJ			
SOT343R						97-05-21

Figure 11.6: *SOT*343*R* package description

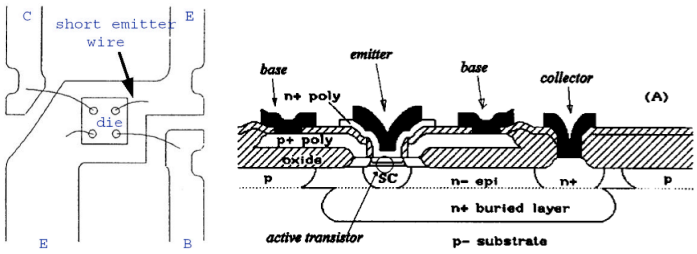


Figure 11.7: die connection if the fifth generation transistor from philips

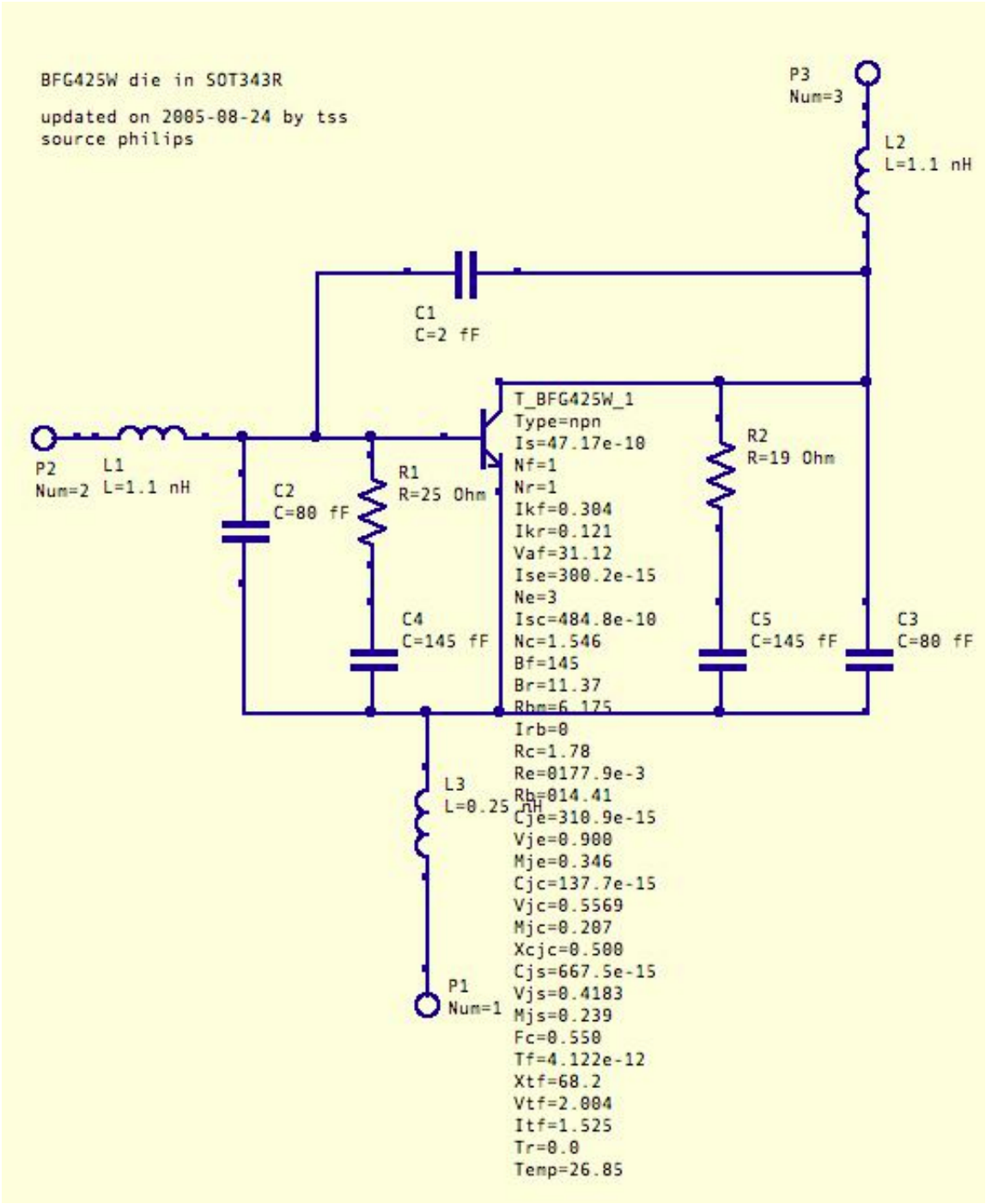


Figure 11.8: $bfg425W$ in $sot343R$ package description

11.5 small signal S parameter verification

In this section we will need to redraw a new schematics using the model we have created, plus some extra components to place the measurements ports [2].
You should have a schematics like the one mentionned in fig11.9.

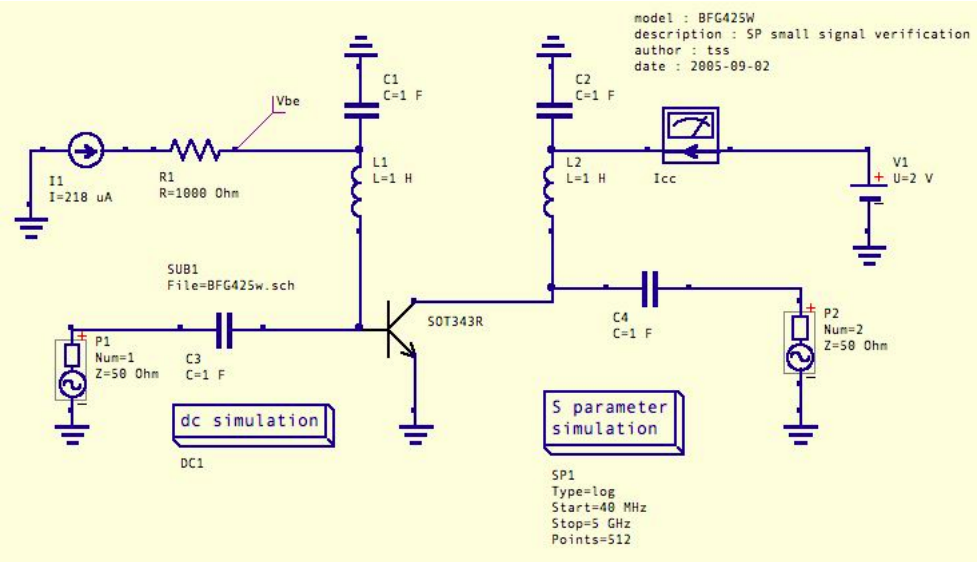


Figure 11.9: schematics used for S parameters simulation

The components used to verify the model could be strange (inductor of $1H$ and capacitor of $1F$) It is normal since we need to have a very wide band response on the circuit, and since we want to caracterize only the active device, and compare with the datasheet. An other way is to use DC bloc or DC feed or bias Tee to provide the power supply to the component. This is the right way to do it.
you should then create a display to visualize the S parameters : generally s_{11} and s_{22} are in the smith and s_{12} and s_{21} are in polar
We have now to compare these results with the measured parameters from philips :

```
! Filename: 225bfg425.001
! BFG425W Field C1
! V1=8.667E-001V,V2=2.000E+000V, I1=3.585E-004A, I2=2.496E-002A
!                S11              S21              S12              S22
!Freq(GHz)  Mag     Ang      Mag     Ang      Mag     Ang      Mag     Ang
# GHz   S   MA   R 50
   0.040    0.325  -8.696   38.472 173.381    0.002  71.865    0.923  -3.072
   0.100    0.331 -23.004   37.457 164.549    0.005  83.280    0.915  -9.551
```

[2]We will another method when we will use the device in a real project

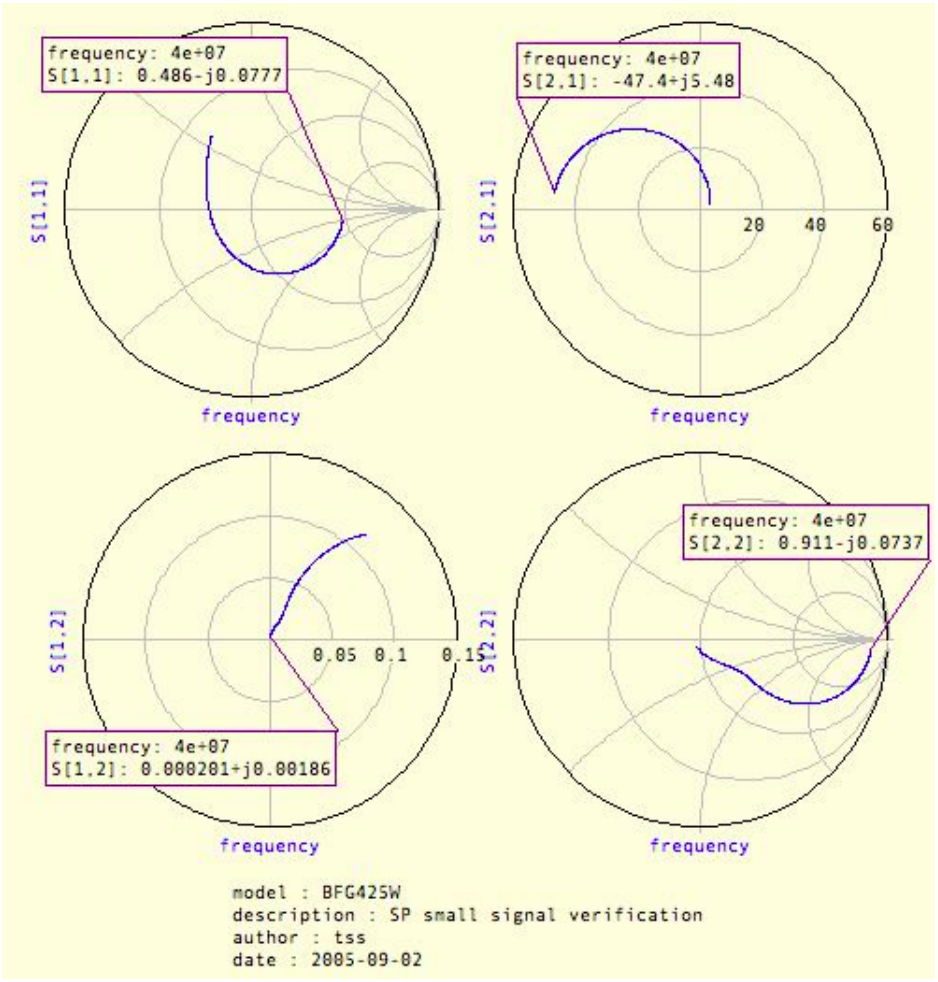


Figure 11.10: S parameters simulation for model verification

```
  0.200    0.315 -44.455   34.771 150.487    0.008  75.947    0.863 -18.965
  0.300    0.296 -63.008   31.364 138.811    0.012  71.608    0.794 -26.449
  0.400    0.278 -79.654   27.951 128.829    0.015  68.186    0.725 -32.076
  0.500    0.265 -94.339   24.856 120.248    0.017  65.974    0.664 -36.332
  0.600    0.254 -106.508   22.159 113.362    0.020  64.514    0.613 -39.533
  0.700    0.246 -116.820   19.885 107.530    0.022  63.362    0.569 -42.071
  0.800    0.240 -126.472   17.964 102.255    0.024  62.701    0.533 -44.121
  0.900    0.235 -134.500   16.345  97.645    0.027  61.910    0.504 -45.968
  1.000    0.232 -141.743   14.958  93.487    0.029  61.280    0.479 -47.614
  1.100    0.230 -148.265   13.770  89.661    0.031  60.570    0.457 -49.172
  1.200    0.230 -154.216   12.748  86.091    0.033  59.878    0.438 -50.696
  1.300    0.230 -159.761   11.850  82.773    0.036  59.238    0.421 -52.103
  1.400    0.231 -164.776   11.070  79.671    0.038  58.509    0.406 -53.483
  1.500    0.233 -169.782   10.383  76.687    0.040  57.719    0.392 -54.842
  1.600    0.234 -174.382    9.766  73.821    0.043  56.846    0.380 -56.285
  1.700    0.236 -178.496    9.213  71.086    0.045  56.001    0.369 -57.740
  1.800    0.238 177.334    8.725  68.404    0.047  54.999    0.358 -59.199
  1.900    0.241 173.487    8.277  65.836    0.050  53.983    0.348 -60.790
  2.000    0.244 169.856    7.874  63.295    0.052  52.923    0.338 -62.399
  2.200    0.251 162.836    7.172  58.413    0.057  50.729    0.319 -65.657
  2.400    0.259 156.208    6.578  53.682    0.062  48.414    0.301 -68.988
  2.600    0.268 150.081    6.068  49.042    0.067  45.958    0.283 -72.558
  2.800    0.277 144.221    5.628  44.575    0.072  43.380    0.266 -76.167
  3.000    0.288 138.650    5.244  40.174    0.077  40.713    0.248 -80.054
  3.500    0.319 125.843    4.470  29.452    0.090  33.634    0.204 -90.648
  4.000    0.352 113.999    3.873  18.944    0.102  26.177    0.158 -103.541
  4.500    0.389 103.406    3.406   8.713    0.113  18.415    0.113 -121.590
  5.000    0.431  92.903    3.011  -1.792    0.123   9.782    0.071 -156.899
  5.500    0.463  82.559    2.658 -11.364    0.131   2.534    0.054 148.652
  6.000    0.506  73.164    2.374 -21.684    0.138  -6.413    0.095 100.575
  6.500    0.516  66.705    2.179 -28.681    0.152 -10.089    0.112  92.309
  7.000    0.551  59.664    2.011 -37.894    0.164 -17.920    0.164  82.321
  7.500    0.610  50.773    1.808 -49.313    0.166 -29.630    0.246  65.957
  8.000    0.644  43.502    1.653 -58.585    0.172 -37.580    0.300  56.971
  8.500    0.683  35.816    1.496 -68.478    0.175 -46.984    0.361  47.167
  9.000    0.709  27.972    1.338 -77.310    0.173 -55.176    0.412  37.289
  9.500    0.736  20.858    1.212 -85.841    0.172 -63.448    0.449  29.117
 10.000    0.764  14.187    1.105 -95.600    0.173 -72.751    0.505  22.602
 10.500    0.785   7.330    0.997 -104.961    0.171 -81.774    0.554  14.956
 11.000    0.802   0.219    0.884 -113.744    0.164 -91.275    0.593   6.422
 11.500    0.815  -6.751    0.791 -122.965    0.158 -100.952    0.631  -0.521
 12.000    0.822 -13.843    0.690 -131.882    0.149 -111.108    0.667  -8.548
!     DEEMBEDDED NOISE DATA
```

```
!FREQUENCY     FMIN     GAMMA OPT         Rn
!   (GHz)       (dB)     Mag     Ang    (NORMALIZED)
```

Using these parameter, we shoul compare on the sample display the modelised results and the measurements results, or directly show the error using equations. First we compare the results.

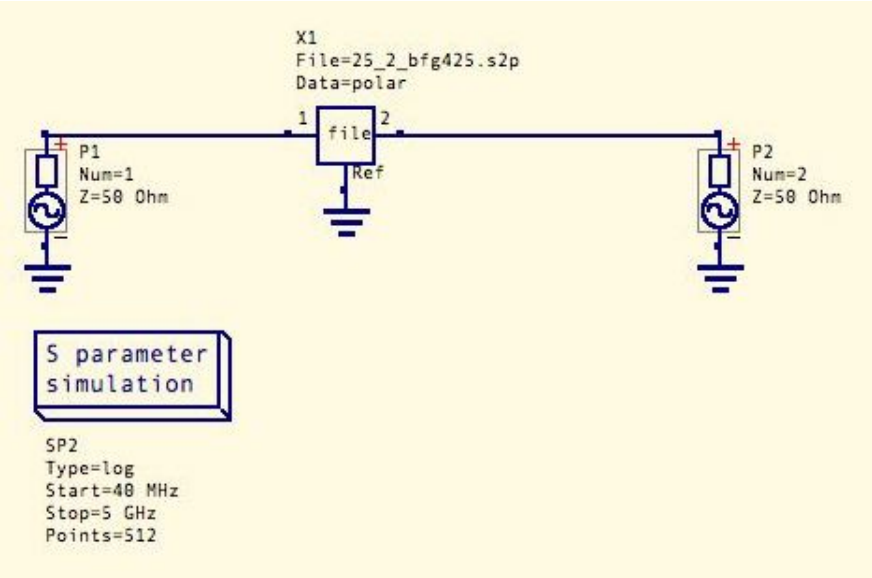


Figure 11.11: schematics used for S parameters from manufacturer

In the display that is used for the S parameters that we have simulated from our modelisation, you can add the results from the meaurements files by adding a measurement of $S_{i,j}$ using the right dataset with the combo box. You should obtain the difference between the two.
By doing this, you should obtain the results presented in the figure 11.12.

IMPORTANT NOTE : The differences, you should obtain are still on investigation for now.

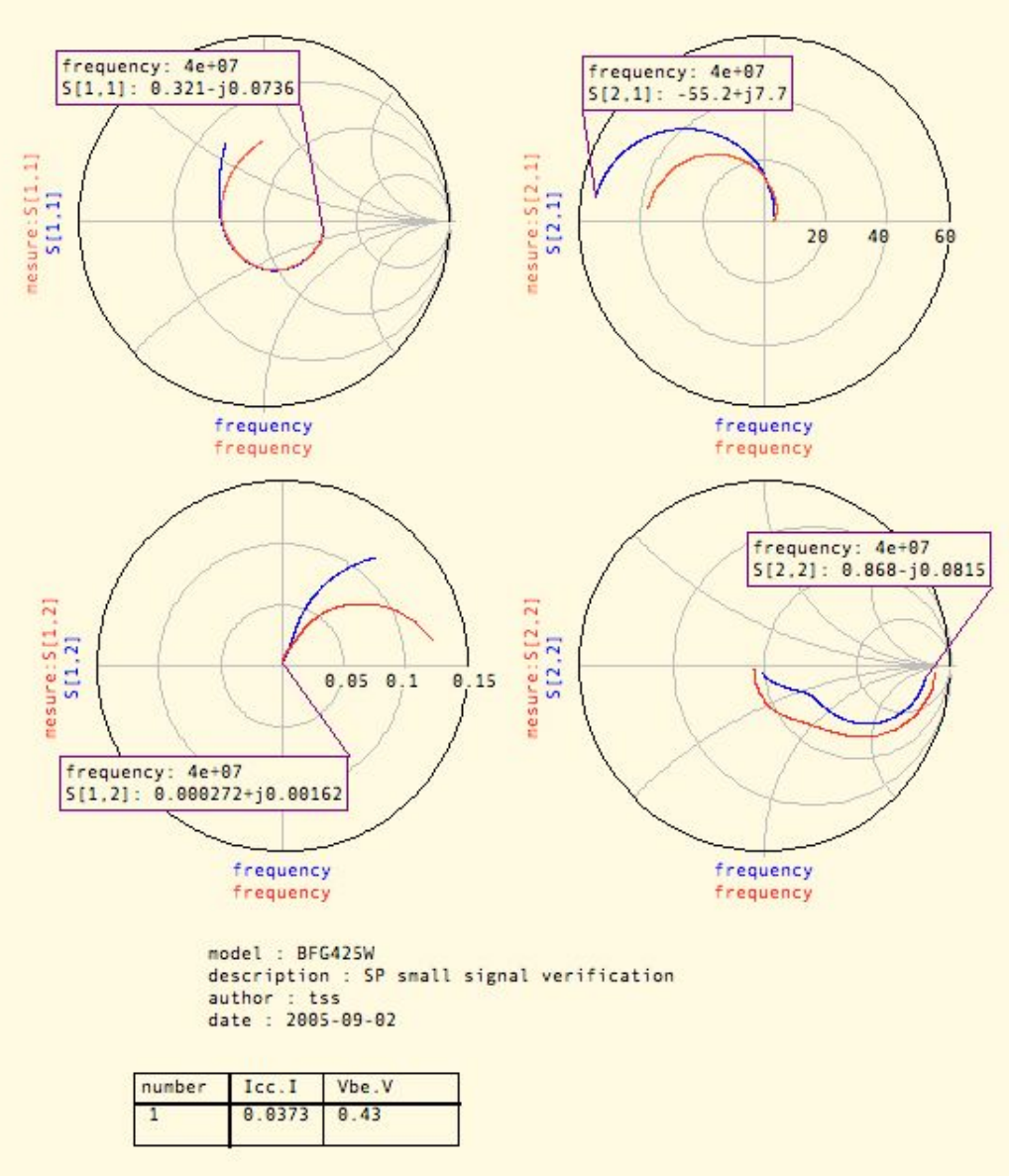


Figure 11.12: Results from model and from meaures compared together

12 Power Amplifier Design

warning

This chapter will describe an RF design issue using QUCS. The author assume that the basic manipulation of qucs is known. You will find herein mainly a MacOsX description that is close to a linux or unices architecture.

12.1 Field of interest

This power amplifier will be used in a more complex system taht I can not describe herein, but the application is inside the $868MHz$ ISM frequency band. This amplifier is considered as power amplifier since it is not a LNA, but its power is not very high as well as you can see in the following system specification. It is more a low input power amplifier driving relatively low current. An application note with really high power level such several watts will be an other chapter.

12.2 System consideration

As a system point of view we need first to specify what kind of function we need. this function will be defined as mentionned in table 12.1.

Table 12.1: System specification for the design of a PA

parameter	description	min	typ	max	unit
F_o	frequency of operation	863	868.6	870	MHz
Icc	current consumption		20	25	mA
Z_{in}	input impedance		50		Ω
Z_{out}	output impedance		50		Ω
P_{in}	input power	−15	−10	−8	dBm
P_{out}	input power	5	10	12	dBm
V_{cc}	DC supply voltage	2.45	2.5	2.55	V

Cost issue is very important, therefore only one active component is allowed, and the BOM [1] should be reduced as much as possible.

This design should work on a FR4 PCB used in a production line. The parameters of such substrate is quite uncontrolled but can be caracterized, as long as you keep the same supplier (avoid strange suppliers who can change the FR4 composition without notice). As mentionned previously you can describe a substrate inside the library with the following lines :

```
<SUBST FR4_ 1 0 0 -30 24 0 0
"4.7" 1 "0.7 mm" 1 "35 um" 1 "2e-4" 1 "0.022e-6" 1 "0.15e-6" 1
>
```

The height of the substrate is 0.7mm but this describe only one RF layer of the full implementation of the circuit which is a four layour board. The two inner layer are power and ground, the top and bottom layer are RF layers.

12.3 Biasing consideration

In this section we will see how the biasing is made, especially using a emitter feed back technic. If you remember well the data sheet of the transistor, there is a huge dispersion on the h_{FE}, and some other dispersion have to be taken into account : resistance, supply voltage, ...
The used schematics is shown is fig 12.1. But we need to evaluate the component first. Using small calculus it is easy to figure out the different resistance :
assuming that

$$I_c = \beta I_b \tag{12.1}$$

$$I_{biasBridge} \ll I_b \tag{12.2}$$

$$I_{biasBridge} = \frac{I_c}{10} \tag{12.3}$$

$$R_e = \frac{V_{cc} - V_{ce}}{I_c} \tag{12.4}$$

$$R_1 + R_2 = \frac{10 \times V_{cc}}{I_c} \tag{12.5}$$

$$R_2 = \frac{10}{I_c} \times (V_{cc} - V_{ce} + V_{be}) \tag{12.6}$$

The inputs are :

[1] Bill Of Material

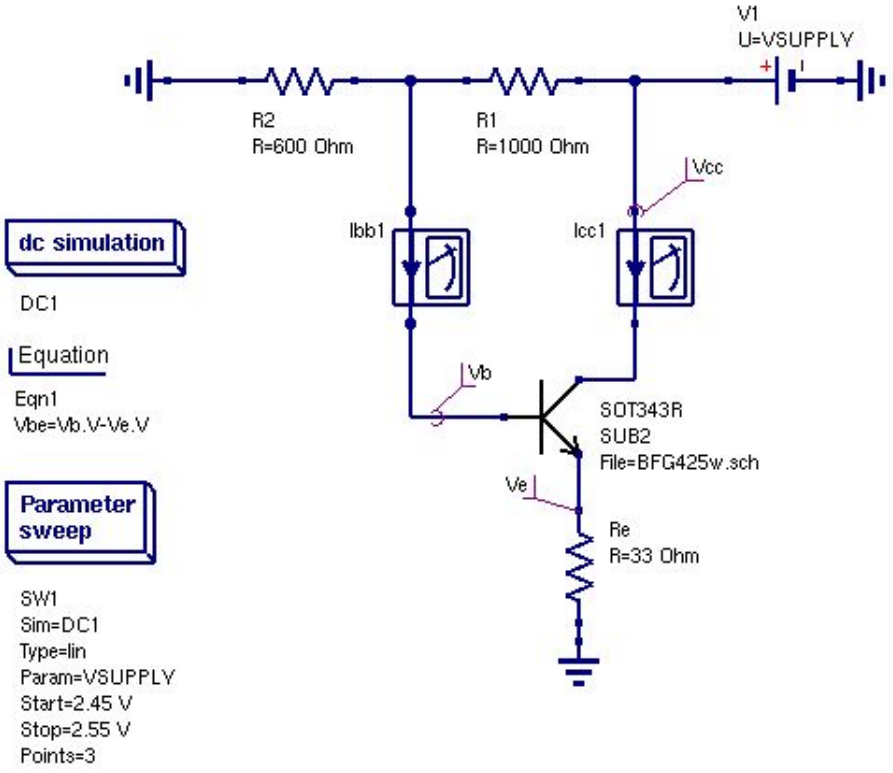


Figure 12.1: Schematics used for this study

- $V_{cc} = 2.5V$
- $V_{be} = 0.412V$
- $I_c = 15mA$

the results are :

- $R_1 = 1K\Omega$
- $R_2 = 600\Omega$
- $R_e = 33\Omega$

Using these values on the schematics, we can now see the stability of the design. Adding the fact that the voltage regulator used in this case has an ondulation of 5 mV in the working domain. You need to simulate the DC schematics by modifying the BF parameter of the transistor from 50 to 120 (since this feature is not enabled in the current version of Qucs 0.0.7).

Table 12.2: Variation of I_c in mA, due to the V_{cc} and β

Vcc vs β	50	80	120
2.45	12.21	13.34	14.07
2.50	12.62	13.78	14.54
2.55	13.03	14.23	15.01

From this table we can extract some stability factor :

$$\frac{\Delta I_{cc}}{\Delta V}|_{\beta=80} = 8.9\mu A/mV \tag{12.7}$$

$$\frac{\Delta I_{cc}}{\Delta \beta}|_{V_{cc}=2.5} = 30\mu A \tag{12.8}$$

$$\frac{\Delta I_{cc}}{\Delta T}|_{\beta=seenote,V_{cc}=2.5} = \ldots \mu A/C \tag{12.9}$$

Note : For the temperature dependance, we need to take the minimum β for the minimum temperature, and the maximum β for the maximum temperature.

12.4 Why thermal design ?

The objective of the thermal design in electronic equipment is to provide as low a temperature rise, ΔT, above ambiant as is practical for a product's electronic components.
As a practical matter, a small 3C to 5C component temperature rise is almost unavoidable, and actually has been found to be desirable. If the rise is less than that, there can be more moistrure-related problems, particularly corrosion and electrical leakage currents.

- Improves performance : avoids calibration drift, maintains phase lock loops, stabilizes gain, ...
- Improves reliability : failure mechanisms accelerate rapidly at higher temperatures through metal migration, increased ion mobility, ...

  In most electronic components, the failure rate doubles for a 10C to 15C rise in temperature and the slope is exponential ! temperature cycling is even worse.

  Temperature rise is particularly hard on components which depend on an internal liquid, such as electrolytic capacitor, batteries, and lubricated bearings.

  Sophisticated thermal design is becoming a necessity as devices becomes smaller and poxer density increase. Examples : VLSICs and surface mount technology SMT.
- Improves life : higher ΔT increases mechanical stress, failures of connections, metalisation contacts,...

12.4.1 Thermal management

The objective of thermal management is to design the internal thermal environment of the electronic equipment so the equipment performance will meet customer expectations. Within the range of environmental conditions where the equipment is expected to operate, the equipment should perform within specifications and operate reliably. In general, the designer has little control over the external environment, so he must design for an anticipated range. He does have more control over the internal environment, but his attention should be directed toward the ultimate goal ; maintaining a suitable environment for the critical components.
Analysis of the thermal environment can usually be divided into several distinct parts because of almost–isothermal boundaries. Consider the typical enclosure system, the isothermal boundaries are :

- the enclosure at T_e
- the interior at T_b
- the component at T_c

Because of these boundaries, ΔT_{jc}, ΔT_{ca} and ΔT_{ja} can be solved independently. ΔT_{ae} and $\Delta T_{e\infty}$ can also be solved independently for a sealed enclosure, but are inter–dependent for a vented or forced air cooled enclosure.

approching the problem During the definition stage of a product, the choice of enclosure is sometimes dictated by a competitor, the customer, or marketing. Frequently the choice is "as small as possible", thus unwittingly passing judgment on a particular choice, it is possible to make a thermal analysis of the proposed enclosure. If the environment created for the component is unsuitable, then additional cooling mechanisms must be developped.
One approch is to simplify the problem to one dimensionnal analysis. Heat energy sources azre assumed to be evenly distributed throughout the volume. The enclosure surface is assumed to be isothermal. The enclosure is assuemd to made of a perfect thermal conductor. (unfortunately, enclosures are more and more being made of plastic, a thermal insulator, which complicates this sample approch).
The external environment is considered to be the walls of a large room of surface emissivity , ϵ , of 1.0 at the same temperature, T_∞, as the surrounding air, and is capable of absorbing an infinite amount of heat energy.
Heat transfert by conduction, radiation, free convection, venting, and forced convection are basically representated by the equation :

$$Q_t = Q_k + Q_r + Q_c + Q_v + Q_f \tag{12.10}$$

The most elusive component, thermal resistance Θ_x, can vary from simple to very complex. Fortunately, most electronic enclosures do not have more than three cooling paths and in many cases, the third path is minor one that can be neglected for ease of calculation.
The following are some generally accepted guidelines that can be used to quickly evaluate a design or configuration. These were obtained from notes provided by [**?**].
Maximum power density :

- for small painted uniformly heated sealed enclosure
  - naturally cooled $< 4mW/cm^3$
  - taller than $60cm < 2mW/cm^3$
- for naturally cooled printed circuit boards $< 16mW/cm^2$
- for forced air cooled printed circuit board $< 110mW/cm^2$
- for small ($60cm$ or less) induced draft cooled enclosure $< 20mW/cm^3$

forced air velocities :

- for PCB cages $> 4m/sec$
- for enclosures $< 7.6m/sec$

12.5 DC Power dissipation

An important issue in power amplifier design is the power dissipation. Even if in this particular case the power dissipation is not that obvious, it is nice to see how we can handle this anyway.

As a student you always learn that you can apply kirchoff law on temperature. This only thing you have to know is the correspondance :

The temperature : is equivalent to the voltage

The power : is equivalent to the current

The thermal resistance : is equivalent to the resistance

You can also take into account some calorific capacity, and perturbation from near effect due to the presence of other source of heating, in a dynamic design, but we will only see the DC power dissipation here ... from this start point you can then imagine whatever you want.

In order to proceed, we need to create a model for this power dissipation. This model can be very simple on its comprehension but very complex since all the parameters are not well known. Therefore we will need to reduce the level of modelisation that is used.
Here are the input parameters :

- The DC power dissipation is $15mA \times 2.5Volts = 37.5mW$
- the thermal resistance of the device is $\theta_{junction_solder} = 350degC/W$
- the thermal resistance of the ambiante is $\theta th_{pcb_air} = 22degC/W$
- the ambiante temperature varies from $-25degC$ to $75degC$ and $25degC$ typical

The schematics used for this simulation is shown is figure 12.2[2].

[2]Note the possiblity to place the results of the simulation directly on the schematics, and some comments on the schematics such as document name, revision, and so on.

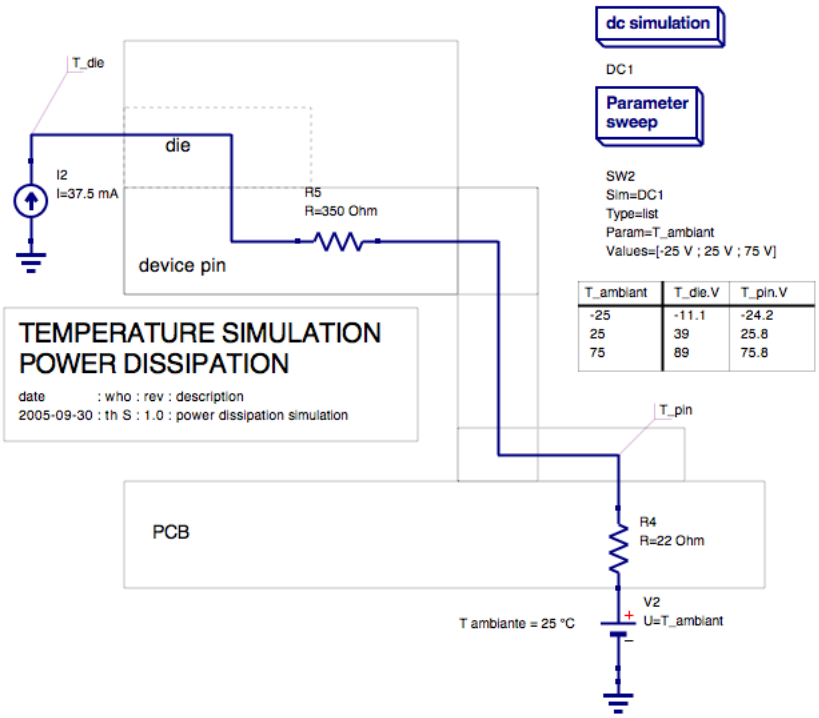


Figure 12.2: Schematics used to simulate the DC power dissipation

12.6 Small signal analysis

The current version of QUCS do not include an Harmonic Balance solver, so we need to do some other simualtions in order to have some ideas on the performances of our design.

13 Low Noise Amplifier Design

This section will describe a two stage LNA. The main goal is to see how we can design this LNA using the QUCS software, but also to find innovative designs for low power [1] solutions.

The main difference between as you should know, between PA and LNA, is that in the design of a LNA the noise factor is crucial, and therefore a trade off has to be made with the gain design. This design rule is well explained in all RF courses, so I will go straightforward to the solution by explaining the "pie" but not the "recipie" !

As mentionned earlier, a particular attention will be placed on the DC study, since the overall current consumtion is a crucial point, and the noise factor that we could have.

13.0.1 System consideration

As a system point of view we need first to specify what kind of function we need. this function will be defined as mentionned in table 13.1.

Table 13.1: System specification for the design of a LNA

parameter	description	min	typ	max	unit
F_o	frequency of operation	863	868.6	870	MHz
Icc	current consumption		0.5	1	mA
Z_{in}	input impedance		50		Ω
Z_{out}	output impedance		50		Ω
P_{in}	input power	−120	−110	−90	dBm
V_{cc}	DC supply voltage	1.4	1.5	1.6	V

note : for the DC supply voltage, we will have to find the correct ripple that is acceptable on this design in order to be able to specify the voltage regulator and its PSRR regarding the other voltage in the design. To proceed, due to the fact that some functionnalities are still missing on QUCS[2] we will use some workaround for the DC study.

[1] be careful when I usually use the term low power , I mean extremlly low power , below the mA generally
[2] normal it is still in development ...

13.0.2 Choice of transistor

In order to design a LNA, a particular attention has to be put in the this choice. Therefore you will need to have a transistor that is well designed for very small current and for LNA application.
I will use the $BFG403AW$ from philips [3]. This transistor belongs to the 5^{th} generation.
To classify directly the different transistors that could be used, the different version
The parameter are the following :

```
TO BE UPDATED WITH THE CORRECT ONE
.SUBCKT BFG403W 1 2 3
L1   2 5 1.1E-09
L2   1 4 1.1E-09
L3   3 6 0.25E-09
Ccb  4 5 2.0E-15
Cbe  5 6 80.0E-15
Cce  4 6 80.0E-15
Cbpb 5 7 1.45E-13
Cbpc 4 8 1.45E-13
Rsb1 6 7 25
Rsb2 6 8 19
Q1 4 5 6 6 NPN

.MODEL NPN NPN
bla bla bla
bla bla bla
bla bla bla
bla bla bla
bla bla bla
.ENDS
```

In order to perform some simulation we should input this component in the device library as mentionned in the chapter on the BJT modeling, and create the schematics thst uses this device. The parasitic element are the same since the package used is the same as the $BFG425W$.

13.0.3 library creation

The major problem in this design is the fact that the needed current on the LNA is not mentioned in the already measured S parameters from the manufacturer. This is one of the

[3] I do not have any stock option with philips, but they provide quite easily some prototypes and the models of their transistors, further more their strategy is to continue to provide small wideband RF transistor, so why not ?

reasons why, we need specicaly a non linear model to describe the transistor. Of course a preliminary calculus could be done using these regular parameters, but since we need also some other features such as distortion and so on, a non linear model is mandatory.

In order to conduct these test, we need to create a model of our component. To perform this you should create or edit the file that contain all the libraries, this file is stored under

`/usr/local/share/qucs/library/philips_RF_widebande_npn.lib`

You can edit this file with `vi`. You need to add the following line :

```
<Qucs Library 0.0.7 "philips RF wideBand">
...
...
...
<Component BFG403W>
  <Description>
    RF wideband NPN 25GHz
    2V, 3mA, 20dB , 2000MHz
    Manufacturer: Philips Inc.
    NPN complement: BFG403W
    -------------------------
    based on spice parameter from philips
    -------------------------
    sept 2005  thierry
  </Description>
  <Model>
<_BJT T_BFG403W_ 1 480 280 8 -26 0 0 "npn" bla bla bla bla>
  </Model>
</Component>
...
...
...
```

13.0.4 DC study

13.0.5 SP study

13.0.6 Non linearities study

13.0.7 Possible improvement tips

14 Microstrip Design

14.1 10dB Directional Coupler Design

The below pictures shows two parallel conductor strips on a dielectric substrate with a backplane metalization. Both the conductor strips have the width W, the height t and the length l. There is a finite gap S between the conductors. The substrates height is denoted by h. With the gap between the conductor strips small enough a capacitive as well as inductive coupling occurs.

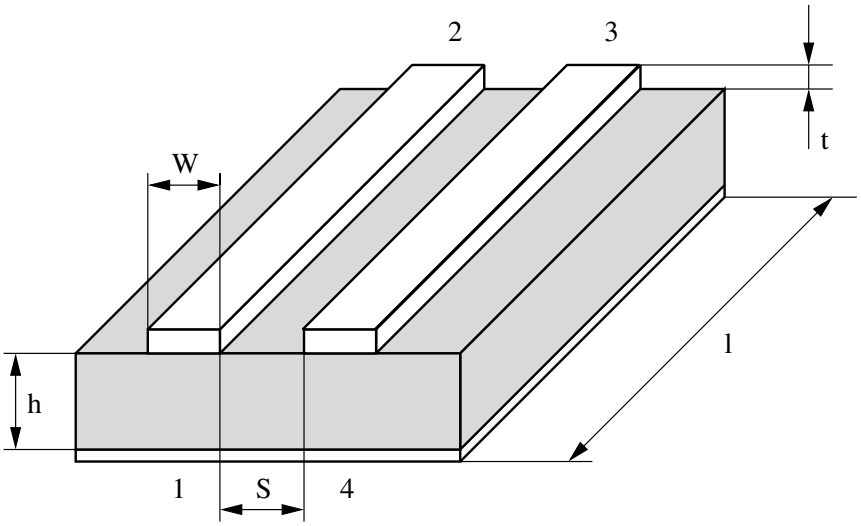


Figure 14.1: microstrip directional coupler

Such a microstrip structure is called "microstrip coupled lines". Also defined in figure 14.1 the port numbers 1...4.

14.1.1 Some boring theory beforehand

There are two types of directional couplers: backward (coupling from port 1 to port 4) and forward (coupling from port 1 to port 3) couplers.

The S-parameters of an ideal directional backward coupler are as follows – with C denoting

the coupling coefficient.

$$\begin{aligned} S_{21} &= \sqrt{1 - C^2} \\ S_{41} &= C \\ S_{31} &= 0 \\ S_{11} = S_{22} = S_{33} = S_{44} &= 0 \end{aligned}$$

In a three conductor system – as the microstrip coupled lines are – there are two types of modes: even and odd. Thus such a system is described by odd and even characteristic impedances ($Z_{L,o}$ and $Z_{L,e}$) and odd and even effective dielectric constants ($\varepsilon_{r,eff,o}$ and $\varepsilon_{r,eff,e}$). The characteristic equations for an ideal backward coupler are

$$\begin{aligned} \varepsilon_{r,eff,e} &= \varepsilon_{r,eff,o} \\ Z_{L,e} &\neq Z_{L,o} \end{aligned}$$

and those for an ideal forward coupler are

$$\begin{aligned} \varepsilon_{r,eff,e} &\neq \varepsilon_{r,eff,o} \\ Z_{L,e} &= Z_{L,o} \end{aligned}$$

The S-parameters of the ideal directional forward coupler are as follows.

$$\begin{aligned} S_{21} &= \sqrt{1 - C^2} \\ S_{31} &= C \\ S_{41} &= 0 \\ S_{11} = S_{22} = S_{33} = S_{44} &= 0 \end{aligned}$$

For both ideal – forward and backward – couplers the reflection coefficients are zero. Port 1 is called the **injection port**. Port 2 is the **transmission port**. In a backward coupler port 4 is the **coupled port** and port 3 is called the **isolated port**. In a forward coupler it's the other way around.

Please note: The given S-parameters for forward and backward couplers are valid for all side termination of each port with the reference impedance Z_L – usually 50Ω.

14.1.2 Design equations

In microwave labs backward line couplers are most wide spread. The basic design equations can be written as

$$\begin{aligned} C &= \frac{Z_{L,e} - Z_{L,o}}{Z_{L,e} + Z_{L,o}} \\ \beta \cdot l &= \frac{\pi}{2} \\ Z_L^2 &= Z_{L,o} \cdot Z_{L,e} \\ Z_{L,e} &= Z_L \cdot \sqrt{\frac{1+C}{1-C}} \\ Z_{L,o} &= Z_L \cdot \sqrt{\frac{1-C}{1+C}} \end{aligned}$$

With

$$\begin{aligned} \beta \cdot l &= \frac{\pi}{2} \\ \rightsquigarrow l &= \frac{\pi}{2 \cdot \beta} = \frac{\pi \cdot c}{2 \cdot \omega} = \frac{c}{4 \cdot f} = \frac{\lambda}{4} \end{aligned}$$

the length l of such a coupler is defined by a quarter wavelength. Both the characteristic impedances can be computed by the reference impedance Z_L, i.e. 50Ω, and the coupling coefficient C.

14.1.3 Applying the design equations

With the previous definitions it's easy to design the 10dB directional backward coupler. We have the reference impedance $Z_L = 50\Omega$ and the coupling coefficient C in dB. First we linearize the coupling coefficient.

$$\begin{aligned} C_{dB} &= -10\text{dB} \\ \rightsquigarrow C &= 10^{C_{dB}/20} = 10^{-0.5} \approx 0.316 \end{aligned}$$

Now we compute the even and odd impedances.

$$\begin{aligned} Z_{L,e} &= Z_L \cdot \sqrt{\frac{1+C}{1-C}} \approx 69.4\Omega \\ Z_{L,o} &= Z_L \cdot \sqrt{\frac{1-C}{1+C}} \approx 36.0\Omega \end{aligned}$$

14.1.4 What next?

All grey theory you may think... With the impedances at hand the engineer had to go into magic diagrams and find physical dimensions of his coupler. But now there is Qucs. Things get easier.

Just select **Tools → Line Calculation** in the menubar or press **Ctrl+3** to start the transmission line calculator.

Then choose **Coupled Microstrip** in the **Transmission Line Type** selection box. Something likely shown in figure 14.2 should appear.

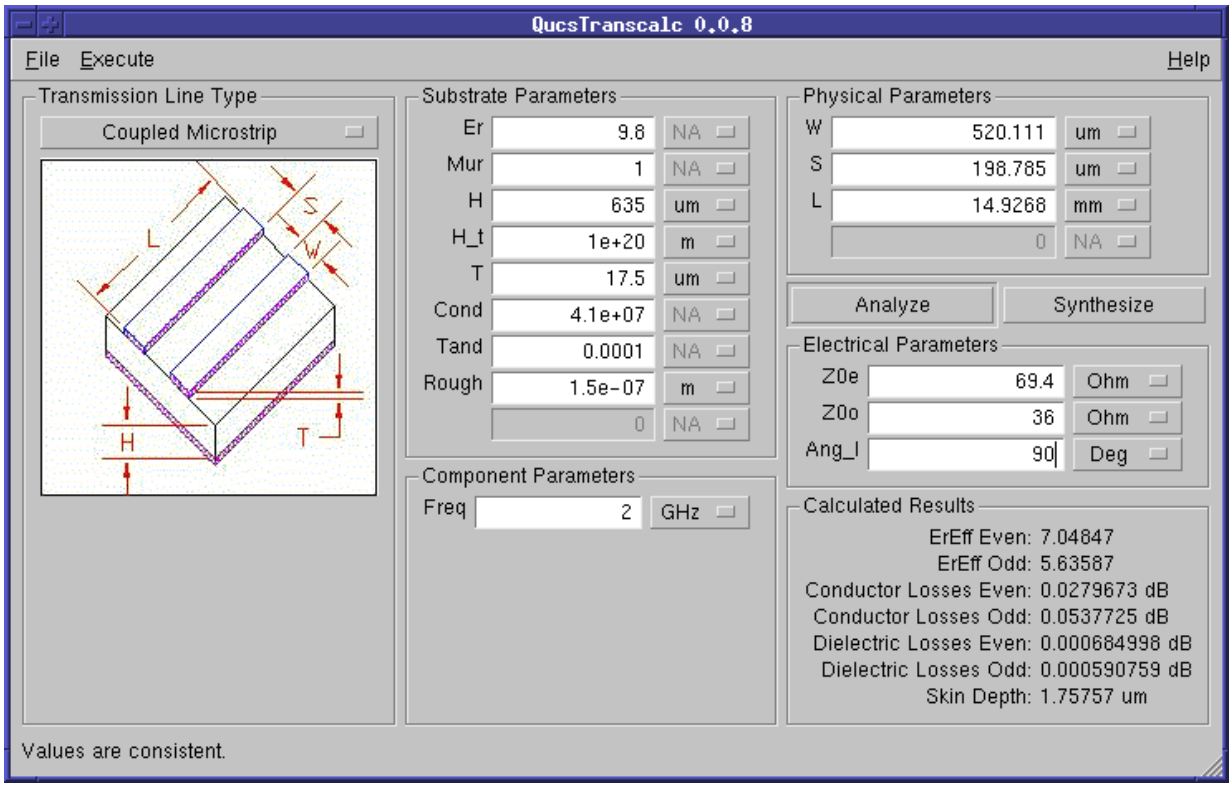


Figure 14.2: Qucs Transcalc screenshot

Type in the calculated **69.4** in the **Z0e** field, **36.0** in the **Z0o** field and **90** in the **Ang_l** field of the **Electrical Parameters** panel. The **Ang_l** field denotes the desired electrical length of the line (remember: $90° \simeq \pi/2$). Choose the **Deg** unit.

Our selected design frequency is 2GHz. Thus type in this value in the **Freq** field of the **Component Parameters** panel.

Then press the **Synthesize** button or press **F4**. The program calculates the physical parameters **W**, **S** and **L** in the **Physical Parameters** panel.

Please note: Depending on the substrate (shown in the **Substrate Parameters** panel) the calculated values may vary.

Finally we got

$$W = 520\mu\text{m}$$
$$S = 199\mu\text{m}$$
$$L = 14.93\text{mm}$$

All done with designing... Feel any better?

14.1.5 Verification of the design

Ok. Let's verify what we have designed so far. Choose **Execute → Copy to Clipboard** from the menubar or press **F2**. This copies the currently shown microstrip coupled line in Qucs Transcalc into the global clipboard.

Now switch to an empty Qucs schematic and press **Ctrl+V**. This inserts the previously entered clipboard content – and click with the left mouse button in order to place the selection into the schematic. This should give you something likely shown in figure 14.3.

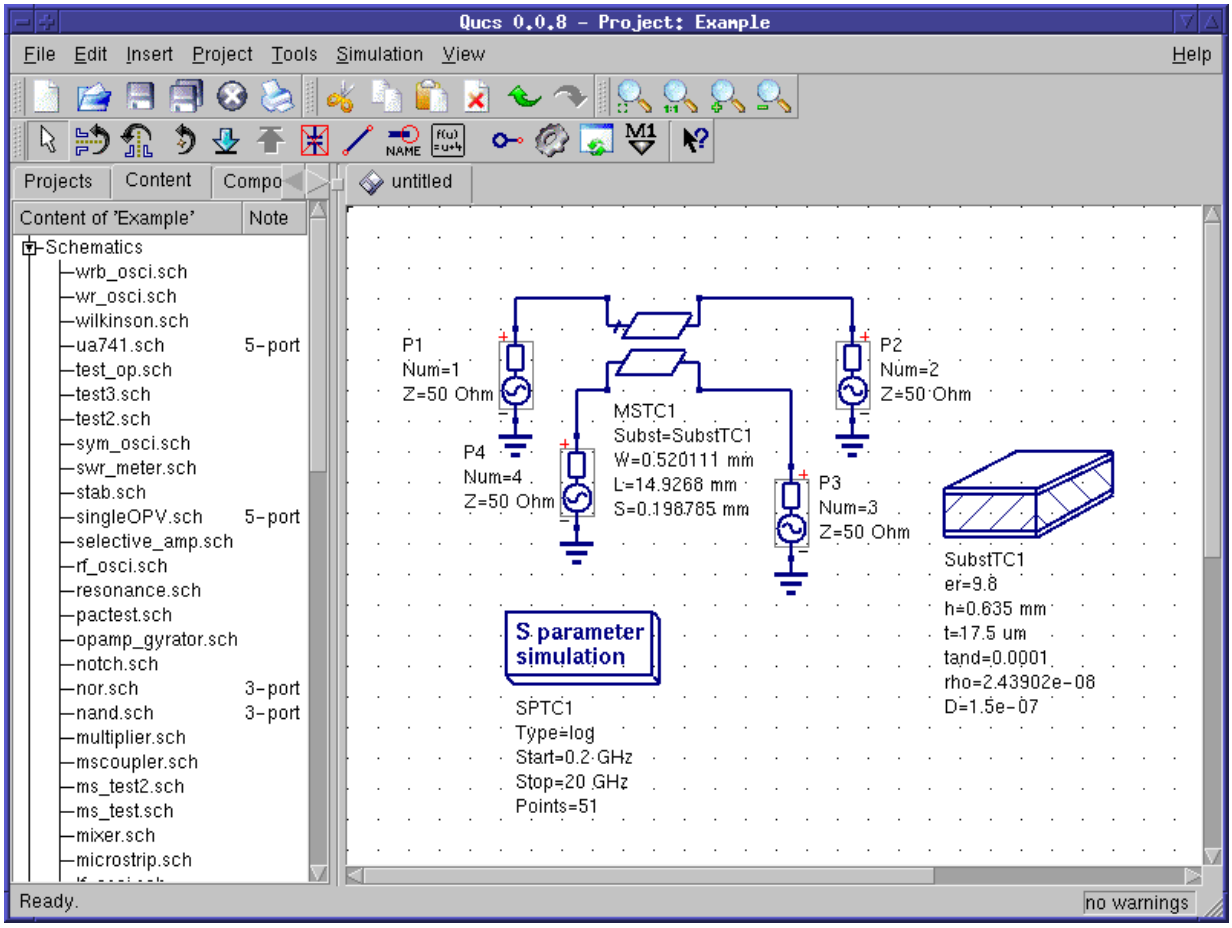


Figure 14.3: coupled microstrip lines in a Qucs schematic

Now press the equation button (shown in figure 14.4) in Qucs's toolbar.

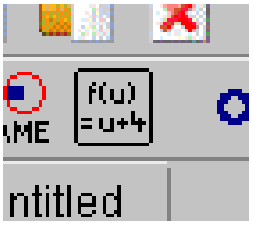


Figure 14.4: equation button

Place the equation into the schematic and enter the following equations. Press **Add** in the equation dialog (see figure 14.5) to add new equations. Finally press the **OK** button.

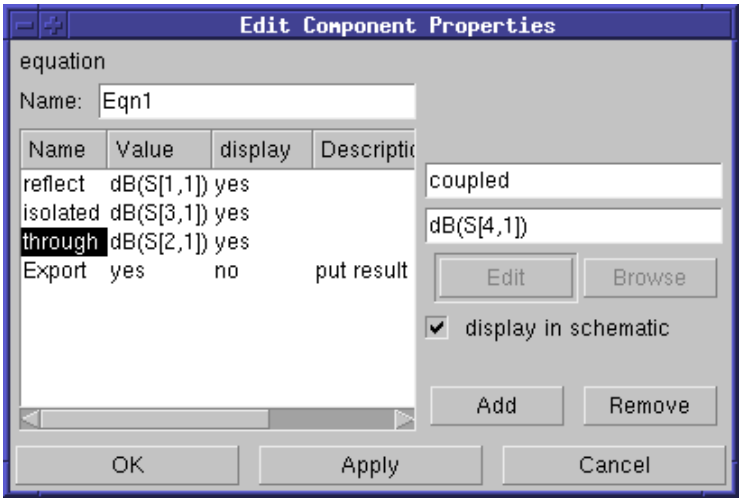


Figure 14.5: equation dialog

Also edit the properties of the **MSTC1** component reducing the number of digits. This will ensure that your technology is able to use these values when (if) they decide to produce your design.

Now edit the S-parameter simulation properties. You can do that either by double clicking the component and use the component dialog. Or you can directly click on the values in the schematic and fill in **0.2 GHz** for **Start**, **4.2 GHz** for **Stop** and **101** for **Points**.

Finally save your schematic by pressing **Ctrl+S**. Check whether all looks like as shown in figure 14.6.

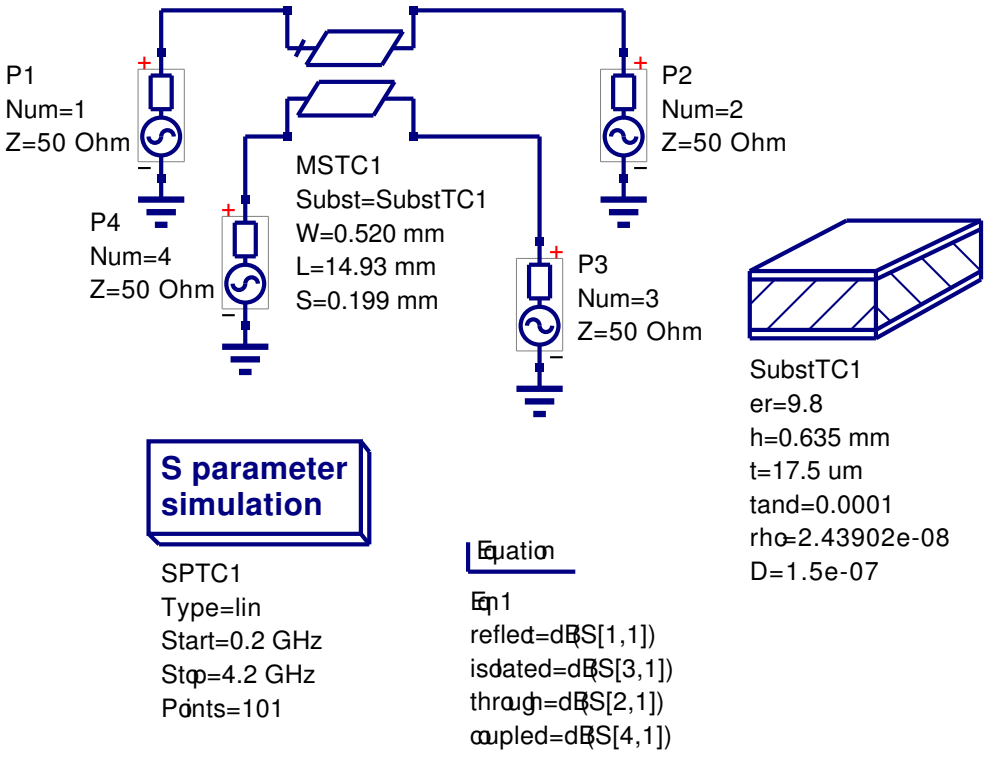


Figure 14.6: final microstrip coupler schematic

Now select **Simulation** → **Simulate** from the menubar or just press **F2** to simulate the schematic.

When the simulation windows disappears then choose a **Cartesian** diagram from the left hand selection view and place the diagram into the (yet empty) data display area. Double click the **through**, **reflect**, **isolated** and **coupled** data items in order to add it to the diagram within the diagram dialog as shown in figure 14.7.

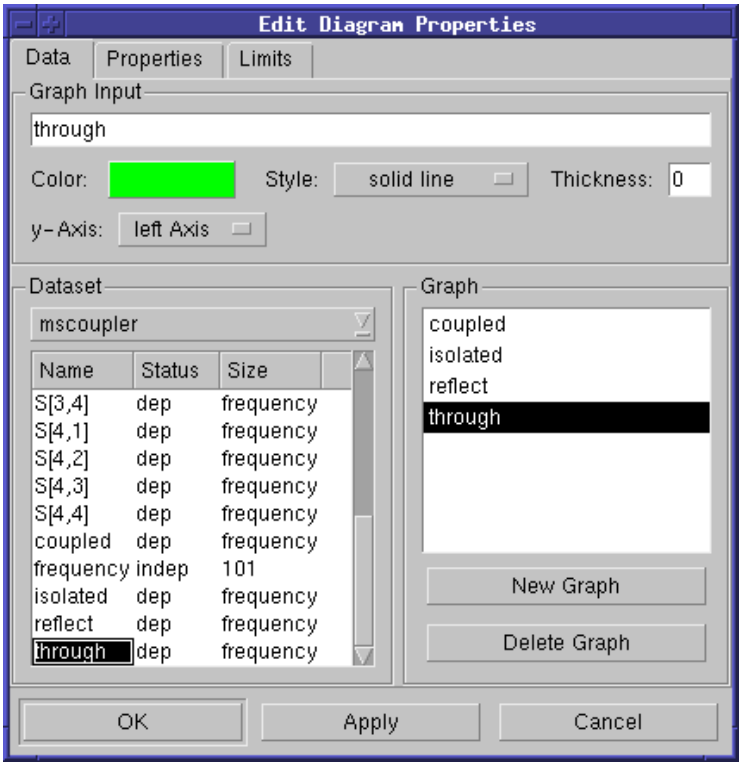


Figure 14.7: diagram dialog

Press **OK** to finish the diagram dialog. Afterwards you will see the following diagram.

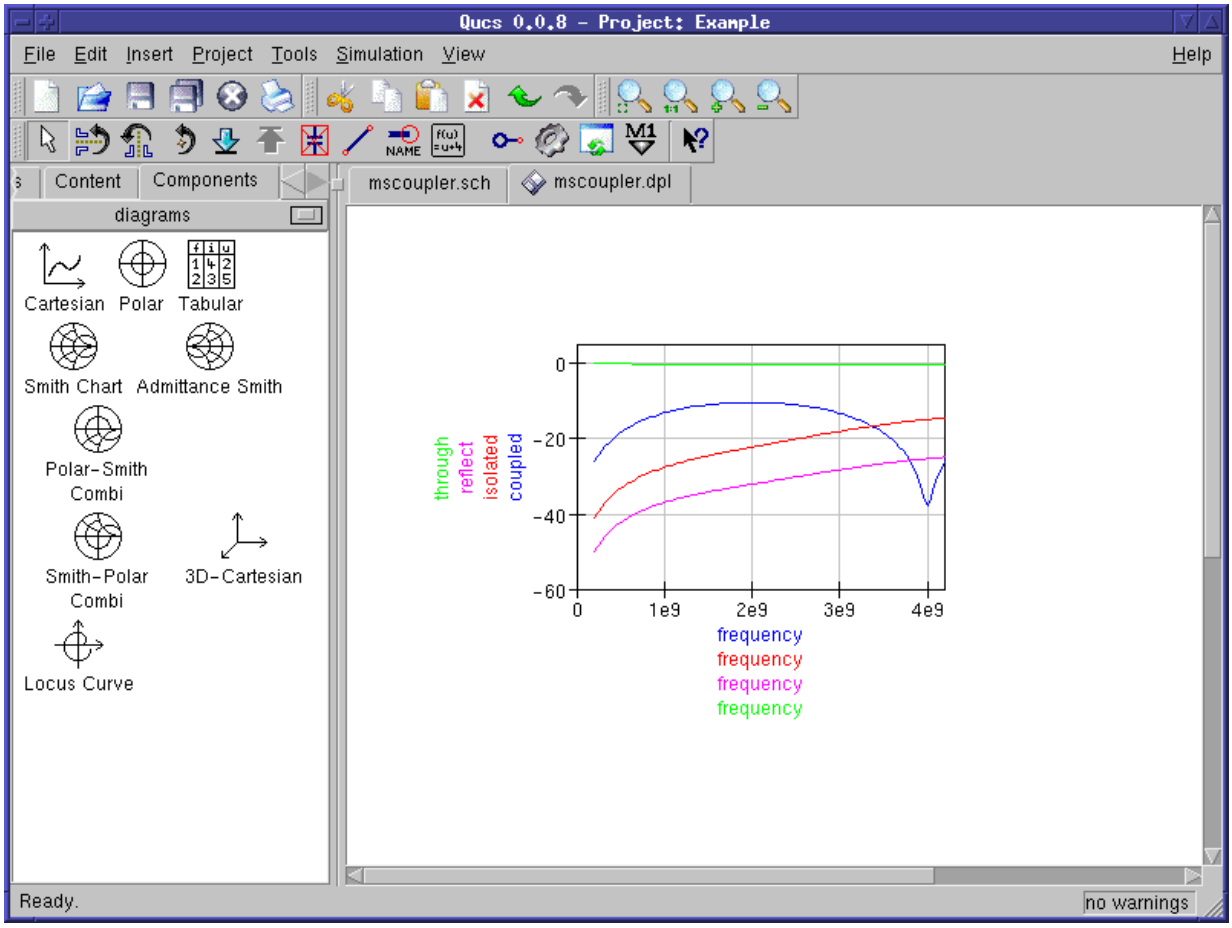


Figure 14.8: microstrip coupler simulation results

14.1.6 Suggested improvements

By use of the diagram dialog (double click the diagram) you may improve[1] the data visualization as you see it fit. I manually fixed the y-axis limits, set markers and set curve thickness to 2 points. Also I entered a common x-axis label. See figure 14.9 how it looks now.

[1]... to feel even better.

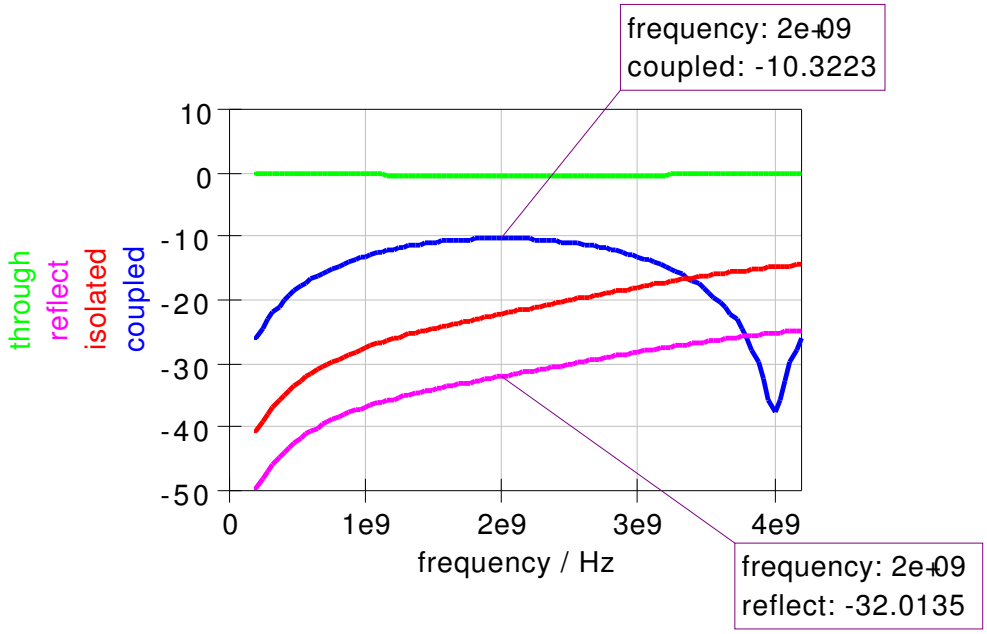


Figure 14.9: directional coupler simulation result diagram

The marker on the **coupled** curve shows a coupling factor of **-10.32** at a frequency of 2GHz (double click marker to change precision of the marker data). This is a bit way off for which we tried to design it for.

Seems like coupling between the lines is a bit too weak. So we reduce the gap between the strip conductors **S** by 16.5µm to be **0.1825 mm** and simulate again.

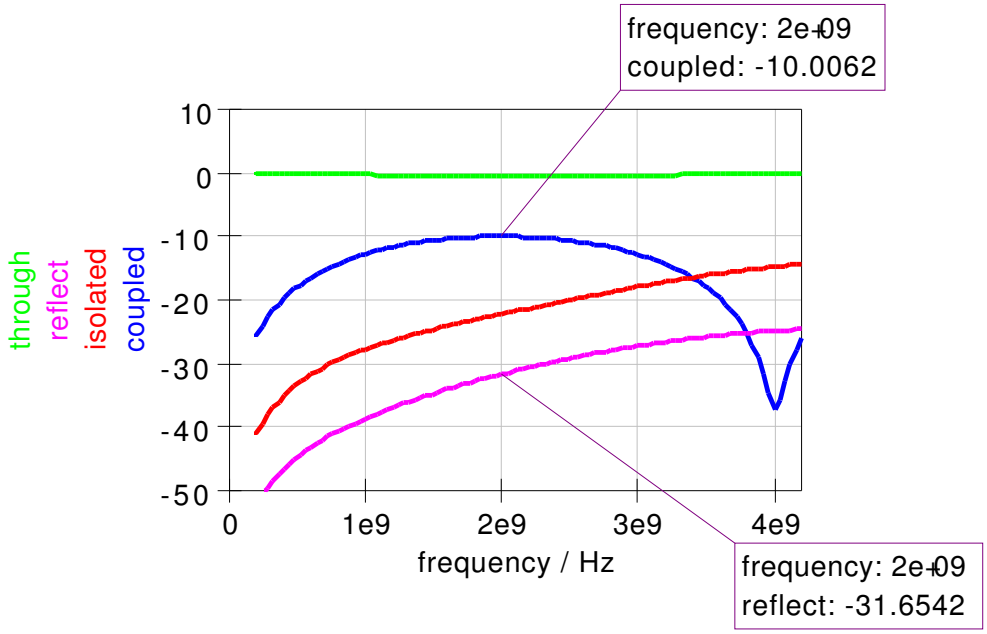


Figure 14.10: optimized directional coupler simulation result diagram

Finally a perfect[2] 10dB coupling as shown in figure 14.10.

14.1.7 Remaining thinkabouts

The diagram in figure 14.10 shows a reflection coefficient of about -31.7dB. The isolation (about -22.2dB) is not as good as planned as well. So – what happened with my design equations?

Have a look at figure 14.2. In the **Calculated Results** panel you see **ErEff Even** and **ErEff Odd** differing significantly which is not what we expect from an ideal backward coupler:

$$\varepsilon_{r,eff,e} = \varepsilon_{r,eff,o}$$

This "problem" arises from the fact that there are two dieletrica involved: air and the substrate. Part of the electromagnetic fields cross air and part of them the substrate. You can inhibit this by a dielectric overlay. It's more expensive to produce but improves your results.

[2]... to feel great.

15 Measurement Expressions Reference Manual

15.1 Introduction

This manual describes the measurement expressions available in ”Qucs”, the ”Quite Universal Circuit Simulator”.

Measurement expressions come into play whenever the results of a ”Qucs” simulation run need post processing. Examples would be the conversion of a simulated voltage waveform from volts to dBV, the root mean square value of that waveform or the determination of the peak voltage. The ”Qucs” measurement functions offer a rich set of data manipulation tools.

If you are not familiar with the way how to enter those formulas, please refer to chapter *“Using Measurement Expressions”*, which points out the possibilities to create and change measurement expressions. Also the data types supported are specified here. Chapter *“Functions Syntax and Overview”* introduces the basic syntax of functions and a categorical list of all functions available. The core of the document, a detailed compilation of all ”Qucs” functions divided into different categories, is presented in chapter *“Math Functions”* and chapter *“Electronics Functions”*. Finally, the *Index* contains an alphabetical list of all functions.

15.2 Using Measurement Expressions

The chapter describes the usage of mathematical expressions for post processing simulation data in “Qucs”, how to enter formulas and modifying them. It gives a brief description of the overall syntax of those expressions.

15.2.1 Entering Measurement Expressions

Measurement expressions generate new datasets by function or operator driven evaluation of simulation results. Those new datasets are accessible in the data display tab after simulation. The related equations can be entered into the schematic editor by the following means:

- Using the equation icon in the “Tools” bar (see fig. 15.1)
- Using menu item “Insert” → ”Insert equation”

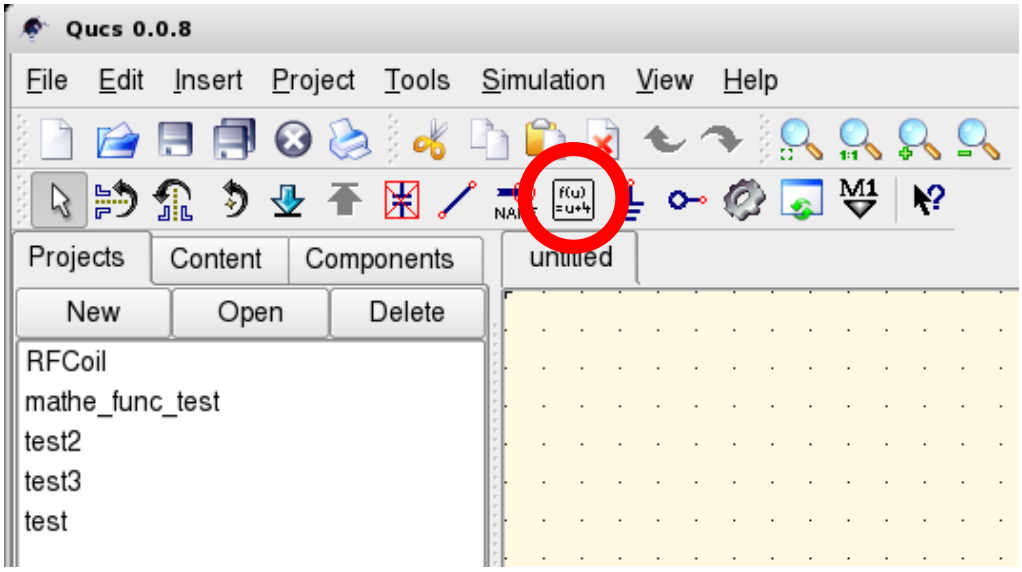


Figure 15.1: Entering a new measurement expression via equation icon

You can now place the equation symbol by mouse click anywhere in the schematic. Each mouse click creates a new equation instance each consisting of a variable number of measurement expressions. Press the Esc key if you do not like further equations.

Another option is to select an existing equation, copy it (either by menu item “Edit” → ”Copy” or by Ctrl + C [1]) and paste it (either by menu item “Edit” → ”Paste” or by Ctrl + V).

After having successfully created an equation instance, you are now able to modify it.

[1] Ctrl + C means that you have to press the Ctrl key and the C key simultaneously.

15.2.2 Changing Measurement Expressions

For sake of simplicity we assume that you have just generated a new equation - if you like to change an existing, more complicated equation the following steps are the same.

Thus, the excerpt of your schematic surface looks like that in fig. 15.2.

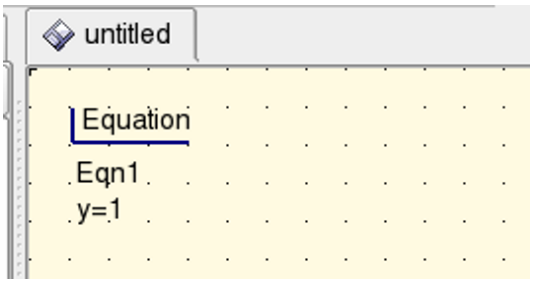


Figure 15.2: Newly created equation

You can now manipulate the current name of the equation instance. Simply click onto "Eqn1", which becomes highlighted. Then type in a new name for it and finalise your inputs with the Enter key.

After that, you can enter a new equation. Again, click onto "y=1". Only the "1" is marked, and you can enter a new expression there. Please use the variables, operators and constants described in chapter *"Syntax of Measurement Expressions"*. Note that you can also refer to results (dependents) of other equations. But how to change the name of the current dependent "y"? Right click onto the equation, and a context menu opens. Select the first item called "Edit properties". A sub window appears, which should look like the one in fig. 15.3. The alternative for entering equations is to double click onto the equation.

You can now change the name of the dependent, the equation itself (which is "1" in the example shown) and the name of the equation. If you do not want the result to be exported into the data display tab, but temporarily need it for further calculations, select "no" in the "Export value" cell.

15.2.3 Syntax of Measurement Expressions

Function names, variable names, and constant names are all case sensitive in measurement expressions - it is distinguished between lowercase and uppercase letters such as 'a' and 'A'.

In functions, commas are used to separate arguments.

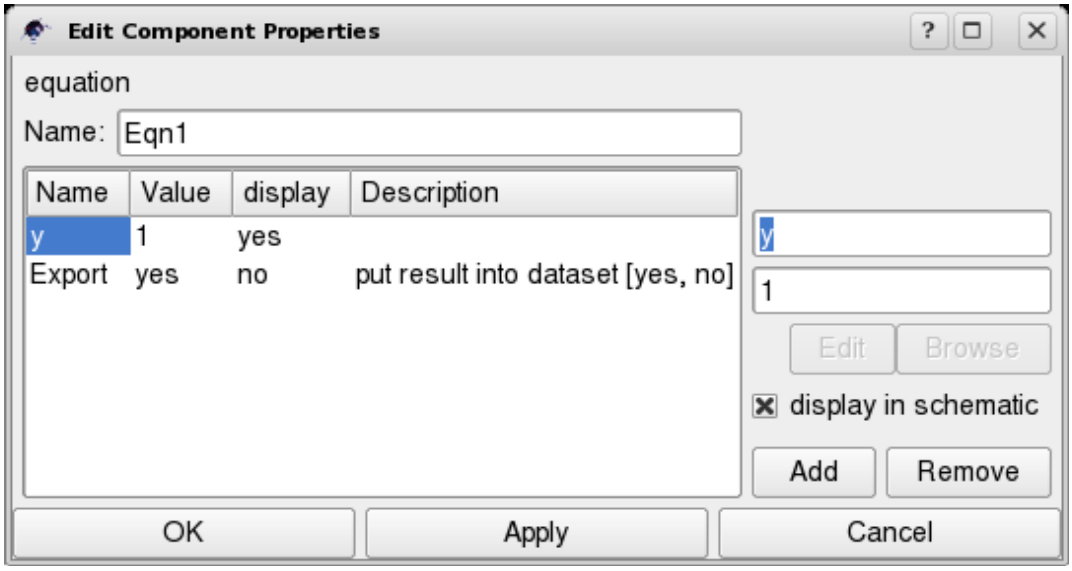


Figure 15.3: Editing equation properties

Variable Names

User defined variable names consist of a letter, followed by any number of letters, digits, or underscores.

The syntax of variable names created by the ”Qucs” simulator is as specified in table 15.1. Please note that all voltages and currents in “Qucs” are peak values except the noise voltages and currents which are RMS values at 1Hz bandwidth.

Numbers

Numbers are written in conventional decimal way, with an optional decimal point between the digits. For powers of ten, the familiar scientific notation with an ’e’ is used. In this way, ’1.234e6’ is an example for the real floating point number 1234000. Imaginary numbers can be entered by a multiplication factor ’i’ or ’j’ (see also table 15.3). An example would be ’1+2*i’ or - if you want to leave out the multiplication sign - ’1+i2’.
Beside the scientific ’e’ notation the following number suffixes can be used (see table 15.2):

Variable Name	Description
nodename.V	DC voltage at node *nodename*
name.I	DC current through circuit component *name*
nodename.v	AC voltage at node *nodename*
name.i	AC current through circuit component *name*
nodename.vn	AC noise voltage at node *nodename*
name.in	AC noise current through circuit component *name*
nodename.Vt	Transient voltage at node *nodename*
name.It	Transient current through circuit component *name*
name.OP	*name* = component name, OP = operating point (device dependent), e.g. D1.Id
S[x,y]	S-parameter, e.g. S[1,1]
Rn	equivalent noise resistance
Sopt	optimal reflection coefficient for minimum noise
Fmin	minimum noise figure
F	noise figure
nodename.Vb	Harmonic balance voltage at node *nodename*

Table 15.1: Syntax of simulator generated variable names

Vectors and Matrices

You can enter vectors and matrices manually by enclosing columns and rows into brackets. Columns are separated by commas, rows by semicolons. A valid matrix entry in a measurement expression would be 'A=[1,2;3,4]', defining the matrix $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$. The notation 'y=[1,2,3,4]' configures the vector $y = \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix}$. You get access to components of matrices and vectors by writing its name followed by brackets. Inside of the latter ranges (see table 15.6) or indices, separated by commas, define the extract you desire. Examples are 'y=M', accessing the whole matrix M, 'y=M[2,3]', extracting the value of the second row and third column of M, or 'y=M[:,3]', obtaining the complete third column.

Built-in Constants

The constants which can be used within measurement expressions are given in table 15.3.

Suffix	Name	Value	Suffix	Name	Value
E	exa	1E+18	m	milli	1E-3
P	peta	1E+15	u	micro	1E-6
T	tera	1E+12	n	nano	1E-9
G	giga	1E+9	p	pico	1E-12
M	mega	1E+6	f	femto	1E-15
k	kilo	1E+3	a	atto	1E-18

Table 15.2: Number Suffixes

Constant	Description	Value
e	Euler's constant	2.718282
i , j	Imaginary unit $\left(\sqrt{-1}\right)$	i1
kB	Boltzmann's constant	1.380658e23 J/K
pi	π	3.141593

Table 15.3: Built-in Constants

Operators

Operator Precedence Expressions are evaluated in the standard way, meaning from left to right, unless there are parentheses. The priority of operators is also handled familiarly, thus for example multiplication has precedence to addition. Tables 15.4 and 15.5 specify sorted lists of all operators, the topmost having highest priority. Operators on the same line have the same precedence.

Operator	Description	Example
()	Parentheses, function call	`y=max(v)`
^	Exponentiation	`y=3^4`
*	Multiplication	`y=3*4`
/	Division	`y=3/4`
%	Modulo	`y=4%3`
+	Addition	`y=3+4`
-	Subtraction	`y=3-4`
:	Range operator	`y=v[3:12]`
+,-	Unary plus, unary minus	`y=+x z=-y`

Table 15.4: Arithmetic Operator Priorities

Operator	Description	Example
()	Parentheses	`a=(x||y)&&z`
!	Negation	`z=!x`
? :	Abbreviation for conditional expression "if x then y else z"	`a=x?y:z`
&&	And	`z=x&&y`
||	Or	`z=x||y`
^^	Exclusive Or	`z=x^^y`
==	Equal	`z=x==y`
!=	Not equal	`z=x!=y`
<	Less than	`z=x<y`
<=	Less than or equal	`z=x<=y`
>	Larger than	`z=x>y`
>=	Larger than or equal	`z=x>=y`

Table 15.5: Logical Operator Priorities

Syntax	Explanation
m:n	Range from index m to index n
:n	Range up to index n
m:	Range starting from index m
:	No range limitations

Table 15.6: Range definition

Ranges The general nomenclature of ranges is displayed in table 15.6. It shows one-dimensional ranges, whereas also n-dimensional ranges are possible, if you consider nested sweeps.

Post Processing of Simulation Data by Expressions

After a simulation has run the results are stored in datasets. Usually, such a dataset is a vector or a matrix, but may also be a real or complex scalar. For transient analysis, this dataset contains voltage or current information over time, for Harmonic Balance it contains amplitudes at dedicated frequencies, while for S-parameter analysis a vector of matrices (thus matrices in dependency of frequency) is returned. In further generalisation the components of vectors and matrices consist of complex numbers.

Additionally, datasets can be generated by using expressions. As an example the linspace() function shall be named, which creates a vector of linearly spaced elements.

15.3 Functions Syntax and Overview

This chapter introduces the basic syntax of the function descriptions and contains a categorical list of all available functions.

15.3.1 Functions Reference Format

"Qucs" provides a rich set of functions, which can be used to generate and display new datasets by function based evaluation of simulation results. Beside a large number of mathematical standard functions such as square root (sqrt), exponential function (exp), absolute value (abs), functions especially useful for calculation and transformation of electronic values are implemented. Examples for the latter would be the conversion from Watts to dBm, the generation of noise circles in an amplifier design, or the conversion from S-parameters to Y-parameters.

Functions Reference Format

In the subsequent two chapters, each function is described using the following structure:

<Function Name>

Outlines briefly the functionality of the function.

Syntax

Defines the general syntax of this function.

Arguments

Name, type, definition range and whether the argument is optional, are tabulated here. In case of an optional parameter the default value is specified. "Type" is a list defining the arguments allowed and may contain the following symbols:

Symbol	Description
$\mathbb{R}$	Real number
$\mathbb{C}$	Complex number
$\mathbb{R}^n$	Vector consisting of n real elements
$\mathbb{C}^n$	Vector consisting of n complex elements
$\mathbb{R}^{m\times n}$	Real matrix consisting of m rows and n columns
$\mathbb{C}^{m\times n}$	Complex matrix consisting of m rows and n columns
$\mathbb{R}^{m\times n\times p}$	Vector of p real $m \times n$ matrices
$\mathbb{C}^{m\times n\times p}$	Vector of p complex $m \times n$ matrices

"Definition range" specifies the allowed range. Each range is introduced by a bracket, either "[" or "]", meaning that the following start value of the range is either included or excluded. The start value is separated from the end value by a comma. Then the end value follows, finished by a bracket again, either "[" or "]". The first bracket mentioned means "excluding the end value", the second means "including".

If a range is given for a complex number, this specifies the real or imaginary value of that number. If a range is given for a real or complex vector or matrix, this specifies the real or imaginary value of each element of that vector or matrix. The symbols mean "includes listed value" and "excludes listed value".

Description

Gives a more detailed description on what the function does and what it returns. In case some background knowledge is presented.

Examples

Shows an application of the function by one or several simple examples.

See also

Shows links to related functions. A mouse click onto the desired link leads to an immediate jump to that function.

15.3.2 Functions Listed by Category

This compilation shows all "Qucs" functions sorted by category (an alphabetical list is given in the appendix). Please click on the desired function to go to its detailed description.

Math Functions

Vectors and Matrices: Creation

eye()	...	Creates n x n identity matrix
linspace()	...	Creates a real vector with linearly spaced components
logspace()	...	Creates a real vector with logarithmically spaced components

Vectors and Matrices: Basic Matrix Functions

adjoint()	...	Adjoint matrix
array()	...	Read out single elements
det()	...	Determinant of a matrix
inverse()	...	Matrix inverse
transpose()	...	Matrix transpose
length()	...	Length of a vector

Elementary Mathematical Functions: Basic Real and Complex Functions

abs()	...	Absolute value
angle()	...	Phase angle in radians of a complex number. Synonym for “arg”
arg()	...	Phase angle in radians of a complex number
conj()	...	Conjugate of a complex number
deg2rad()	...	Converts phase from degrees into radians
hypot()	...	Euclidean distance function
imag()	...	Imaginary value of a complex number
mag()	...	Magnitude of a complex number
norm()	...	Square of the absolute value of a vector
phase()	...	Phase angle in degrees of a complex number
polar()	...	Transform from polar coordinates into complex number
rad2deg()	...	Converts phase from degrees into radians
real()	...	Real value of a complex number
signum()	...	Signum function
sign()	...	Sign function
sqr()	...	Square of a number
sqrt()	...	Square root
unwrap()	...	Unwraps a phase vector in radians

Elementary Mathematical Functions: Exponential and Logarithmic Functions

exp()	...	Exponential function
limexp()	...	Limited exponential function
log10()	...	Decimal logarithm
log2()	...	Binary logarithm
ln()	...	Natural logarithm (base e)

Elementary Mathematical Functions: Trigonometry

cos()	...	Cosine function
cosec()	...	Cosecant
cot()	...	Cotangent function
sec()	...	Secant
sin()	...	Sine function
tan()	...	Tangent function

Elementary Mathematical Functions: Inverse Trigonometric Functions

arccos()	...	Arc cosine (also known as “inverse cosine”)
arccosec()	...	Arc cosecant (also known as “inverse cosecant”)
arccot()	...	Arc cotangent
arcsec()	...	Arc secant (also known as “inverse secant”)
arcsin()	...	Arc sine (also known as “inverse sine”)
arctan()	...	Arc tangent (also known as “inverse tangent”)

Elementary Mathematical Functions: Hyperbolic Functions

cosh()	...	Hyperbolic cosine
cosech()	...	Hyperbolic cosecant
coth()	...	Hyperbolic cotangent
sech()	...	Hyperbolic secant
sinh()	...	Hyperbolic sine
tanh()	...	Hyperbolic tangent

Elementary Mathematical Functions: Inverse Hyperbolic Functions

arcosh()	...	Hyperbolic area cosine
arcosech()	...	Hyperbolic area cosecant
arcoth()	...	Hyperbolic area cotangent
arsech()	...	Hyperbolic area secant
arsinh()	...	Hyperbolic area sine
artanh()	...	Hyperbolic area tangent

Elementary Mathematical Functions: Rounding

ceil()	...	Round to the next higher integer
fix()	...	Truncate decimal places from real number
floor()	...	Round to the next lower integer
round()	...	Round to nearest integer

Elementary Mathematical Functions: Special Mathematical Functions

besseli0()	...	Modified Bessel function of order zero
besselj()	...	Bessel function of n-th order
bessely()	...	Bessel function of second kind and n-th order
erf()	...	Error function
erfc()	...	Complementary error function
erfinv()	...	Inverse error function
erfcinv()	...	Inverse complementary error function
sinc()	...	Sinc function
step()	...	Step function

Data Analysis: Basic Statistics

avg()	...	Average of vector elements
cumavg()	...	Cumulative average of vector elements
max()	...	Maximum value
min()	...	Minimum value
rms()	...	Root Mean Square of vector elements
runavg()	...	Running average of vector elements
stddev()	...	Standard deviation of vector elements
variance()	...	Variance of vector elements
random()	...	Random number between 0.0 and 1.0
srandom()	...	Set seed for a new series of pseudo-random numbers

Data Analysis: Basic Operation

cumprod()	...	Cumulative product of vector elements
cumsum()	...	Cumulative sum of vector elements
interpolate()	...	Equidistant spline interpolation of data vector
prod()	...	Product of vector elements
sum()	...	Sum of vector elements
xvalue()	...	Returns x-value which is associated with the y-value nearest to a specified y-value in a given vector
yvalue()	...	Returns y-value of a given vector which is located nearest to the specified x-value

Data Analysis: Differentiation and Integration

ddx()	...	Differentiate mathematical expression with respect to a given variable
diff()	...	Differentiate vector with respect to another vector
integrate()	...	Integrate vector

Data Analysis: Signal Processing

dft()	...	Discrete Fourier Transform
fft()	...	Fast Fourier Transform
idft()	...	Inverse Discrete Fourier Transform
ifft()	...	Inverse Fast Fourier Transform
fftshift()	...	Move the frequency 0 to the center of the FFT vector
Time2Freq()	...	Interpreted Discrete Fourier Transform
Freq2Time()	...	Interpreted Inverse Discrete Fourier Transform
kbd()	...	Kaiser-Bessel derived window

Electronics Functions

Unit Conversion

dB()	...	dB value
dbm()	...	Convert voltage to power in dBm
dbm2w()	...	Convert power in dBm to power in Watts
w2dbm()	...	Convert power in Watts to power in dBm

Reflection Coefficients and VSWR

rtoswr()	...	Converts reflection coefficient to voltage standing wave ratio (VSWR)
rtoy()	...	Converts reflection coefficient to admittance
rtoz()	...	Converts reflection coefficient to impedance
ytor()	...	Converts admittance to reflection coefficient
ztor()	...	Converts impedance to reflection coefficient

N-Port Matrix Conversions

stos()	...	Converts S-parameter matrix to S-parameter matrix with different reference impedance(s)
stoy()	...	Converts S-parameter matrix to Y-parameter matrix
stoz()	...	Converts S-parameter matrix to Z-parameter matrix
twoport()	...	Converts a two-port matrix from one representation into another
ytos()	...	Converts Y-parameter matrix to S-parameter matrix
ytoz()	...	Converts Y-parameter matrix to Z-parameter matrix
ztos()	...	Converts Z-parameter matrix to S-parameter matrix
ztoy()	...	Converts Z-parameter matrix to Y-parameter matrix

Amplifiers

GaCircle()	...	Circle(s) with constant available power gain Ga in the source plane
GpCircle()	...	Circle(s) with constant operating power gain Gp in the load plane
Mu()	...	Mu stability factor of a two-port S-parameter matrix
Mu2()	...	Mu' stability factor of a two-port S-parameter matrix
NoiseCircle()	...	Generates circle(s) with constant Noise Figure(s)
PlotVs()	...	Returns a data item based upon vector or matrix vector with dependency on a given vector
Rollet()	...	Rollet stability factor of a two-port S-parameter matrix
StabCircleL()	...	Stability circle in the load plane
StabCircleS()	...	Stability circle in the source plane
StabFactor()	...	Stability factor of a two-port S-parameter matrix. Synonym for Rollet()
StabMeasure()	...	Stability measure B1 of a two-port S-parameter matrix
vt()	...	Thermal voltage for a given temperature in Kelvin

15.4 Math Functions

15.4.1 Vectors and Matrices

Creation

eye()

Creates n x n identity matrix.

Syntax

y=eye(n)

Arguments

Name	Type	Def. Range	Required
n	$\mathbb{N}$	$[1, +\infty[$	$\surd$

Description

This function creates the n x n identity matrix, that is

$$\begin{pmatrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & 0 & \ddots & 0 & \vdots \\ 0 & \cdots & 0 & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 \end{pmatrix}$$

Example

`y=eye(2)` returns $\begin{array}{|c|c|}\hline 1 & 0 \\ \hline 0 & 1 \\ \hline \end{array}$.

See also

linspace()

Creates a real vector with linearly spaced components.

Syntax

y=linspace(xs,xe,n)

Arguments

Name	Type	Def. Range	Required
xs	$\mathbb{R}$	$]-\infty,+\infty[$	$\surd$
xe	$\mathbb{R}$	$]-\infty,+\infty[$	$\surd$
n	$\mathbb{N}$	$[2,+\infty[$	$\surd$

Description

This function creates a real vector with n linearly spaced components. The first component is xs, the last one is xe.

Example

`y=linspace(1,2,3)` returns 1, 1.5, 2.

See also

logspace()

logspace()

Creates a real vector with logarithmically spaced components.

Syntax

y=logspace(xs,xe,n)

Arguments

Name	Type	Def. Range	Required
xs	$\mathbb{R}$	$]-\infty,+\infty[$	$\surd$
xe	$\mathbb{R}$	$]-\infty,+\infty[$	$\surd$
n	$\mathbb{N}$	$[2,+\infty[$	$\surd$

Description

This function creates a real vector with n logarithmically spaced components. The first component is xs, the last one is xe.

Example

`y=logspace(1,2,3)` returns 1, 1.41, 2.

See also

linspace()

adjoint()

Adjoint matrix.

Syntax

Y=adjoint(X)

Arguments

Name	Type	Def. Range	Required
X	$\mathbb{R}^{m\times n}$,$\mathbb{C}^{m\times n}$, $\mathbb{R}^{m\times n\times p}$, $\mathbb{C}^{m\times n\times p}$	$]-\infty,+\infty[$	$\surd$

Description

This function calculates the adjoint matrix Y of a matrix X:

$Y = X^H = (X^*)^T$, where X^* is the complex conjugate matrix of X and X^T is the transposed of the matrix X.

Example

`X=eye(2)*(3+i)` returns

3+j1	0
0	3+j1

. Then,

`Y=adjoint(X)` returns

3-j1	0
0	3-j1

.

See also

transpose(), conj()

array()

Read out single elements.

Syntax

The “array()” function is an implicit command. Thus normally the respective first expression (”preferred”) is used.

Syntax	Preferred	Alternative	Preferred	Alternative
1	y=VM[i,j]	y=array(VM,i,j)		
2	y=M[i,j]	y=array(M,i,j)		
3	y=VM[k]	y=array(VM,k)		
4	y=v[i]	y=array(v,i)	y=v[r]	y=array(v,r)
5	y=v[i,r]	y=array(v,i,r)	y=v[r,j]	y=array(v,r,j)
	y=v[i,j]	y=array(v,i,j)	y=v[r1,r2]	y=array(v,r1,r2)
6	y=s[i]	y=array(s,i)		

Arguments

Name	Type	Def. Range	Required
VM	$\mathbb{R}^{m\times n\times p}$, $\mathbb{C}^{m\times n\times p}$	$]-\infty,+\infty[$	$\surd$(Syntax 1 and 3)
M	$\mathbb{R}^{m\times n}$,$\mathbb{C}^{m\times n}$	$]-\infty,+\infty[$	$\surd$(Syntax 2)
v	$\mathbb{R}^{n}$,$\mathbb{C}^{n}$	$]-\infty,+\infty[$	$\surd$(Syntax 4 and 5)
r, r1, r2	Range$xs:xe$	$0\leq xs\leq n-1$, $xs\leq xe\leq n-1$	$\surd$(Syntax 4 and 5)
i	$\mathbb{N}$	$0\leq i\leq m-1$	$\surd$(Syntax 1, 2, 4, 5, 6)
j	$\mathbb{N}$	$0\leq j\leq n-1$	$\surd$(Syntax 1, 2, 5)
k	$\mathbb{N}$	$0\leq k\leq p-1$	$\surd$(Syntax 3)
s	String	Arbitrary characters	$\surd$(Syntax 6)

Description

This function reads out real or complex vectors of matrices, matrices and vectors or strings. Please refer to the following table for the return values:

Syntax	Argument 1	Argument 2	Argument 3	Result
y=VM[i,j]	$VM = (x_{ijk})$	$i \in \mathbb{N}$	$j \in \mathbb{N}$	Vector $(x_{ij1}, \cdots, x_{ijK})$
y=M[i,j]	$M = (x_{ij})$	$i \in \mathbb{N}$	$j \in \mathbb{N}$	Number x_{ij}
y=VM[k]	$VM = (x_{ijk})$	$k \in \mathbb{N}$		Matrix $\begin{pmatrix} x_{11k} & \cdots & x_{1nk} \\ \vdots & \ddots & \vdots \\ x_{m1k} & \cdots & x_{mnk} \end{pmatrix}$
y=v[i]	$v = (v_i)$	$i \in \mathbb{N}$		Number v_i
y=v[xs:xe]	$v = (v_i)$	$xs, \ldots, xe$		Vector $(v_{xs}, \cdots, v_{xe})$
y=v[i,xs:xe]	$v = (v_i)$	$i \in \mathbb{N}$	$xs, \ldots, xe$	Vector $(v_{xs}, \cdots, v_{xe})$
y=v[xs:xe,j]	$v = (v_i)$	$xs, \ldots, xe$	$xs, \ldots, xe$	Vector $(v_{xs}, \cdots, v_{xe})$
y=v[i,j]	$v = (v_i)$	$i \in \mathbb{N}$	$xs, \ldots, xe$	Vector $(v_{xs}, \cdots, v_{xe})$
y=v[xs1:xe1, xs2:xe2]	$v = (v_i)$	$xs1, \ldots, xe1$	$xs2, \ldots, xe2$	Vector $(v_{xs}, \cdots, v_{xe})$
y=s[i]	$s = (s_i)$	$i \in \mathbb{N}$		Character s_i

Again, v denotes a vector, M a matrix, VM a vector of matrices, s a vector of characters and *xs, xs1, xs2, xe, xe1, xe2* are range limiters.

Example

`v=linspace(1,2,4)` returns 1, 1.33, 1.67, 2. Then,

`y=v[3]` returns 2.

See also

det()

Determinant of a matrix.

Syntax

y=det(X)

Arguments

Name	Type	Def. Range	Required
X	$\mathbb{R}^{n\times n}$,$\mathbb{C}^{n\times n}$, $\mathbb{R}^{m\times n\times p}$, $\mathbb{C}^{m\times n\times p}$	$]-\infty,+\infty[$	$\surd$

Description

This function calculates the determinant of a quadratical n x n matrix X. The result is either a real or a complex number.

Example

`X=eye(2)*3` returns $\begin{array}{|c|c|}\hline 3 & 0 \\ \hline 0 & 3 \\ \hline \end{array}$. Then,

`y=det(X)` returns 9.

See also

eye()

inverse()

Matrix inverse.

Syntax

Y=inverse(X)

Arguments

Name	Type	Def. Range	Required
X	$\mathbb{R}^{n\times n}$,$\mathbb{C}^{n\times n}$, $\mathbb{R}^{m\times n\times p}$, $\mathbb{C}^{m\times n\times p}$	$]-\infty,+\infty[$	$\surd$

Description

This function inverts a quadratical n x n matrix X. The generated inverted matrix Y fulfills the equation

$X \cdot Y = X \cdot X^{-1} = 1$, where "$\cdot$" denotes matrix multiplication and "1" the identity matrix.

The matrix X must be regular, that means that its determinant $\Delta \neq 0$.

Example

`X=eye(2)*3` returns $\begin{array}{|c|c|}\hline 3 & 0 \\ \hline 0 & 3 \\ \hline \end{array}$. Then,

`Y=inverse(X)` returns $\begin{array}{|c|c|}\hline 0.333 & 0 \\ \hline 0 & 0.333 \\ \hline \end{array}$.

See also

transpose(), eye(), det()

transpose()

Matrix transpose.

Syntax

Y=transpose(X)

Arguments

Name	Type	Def. Range	Required
X	$\mathbb{R}^{m\times n}$,$\mathbb{C}^{m\times n}$, $\mathbb{R}^{m\times n\times p}$, $\mathbb{C}^{m\times n\times p}$	$]-\infty,+\infty[$	$\surd$

Description

This function transposes a m x n matrix X, which is equivalent to exchanging rows and columns according to

$$Y = X^T = (x_{ij})^T = (x_{ji}) \text{ with } 1 \leq i \leq m,\ 1 \leq j \leq n$$

The generated matrix Y is a n x m matrix.

Example

`X=eye(2)*3` returns $\begin{array}{|c|c|}\hline 3 & 0 \\ \hline 0 & 3 \\ \hline \end{array}$. Then,

`Y=transpose(X)` returns $\begin{array}{|c|c|}\hline 3 & 0 \\ \hline 0 & 3 \\ \hline \end{array}$.

See also

eye(), inverse()

length()

Length of a vector.

Syntax

y=length(v)

Arguments

Name	Type	Def. Range	Required
v	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$ $]-\infty,+\infty[$	$\sqrt{}$	

Description

This function returns the length of vector v.

Example

`length(linspace(1,2,3))` returns 3.

See also

15.4.2 Elementary Mathematical Functions

Basic Real and Complex Functions

abs()

Absolute value.

Syntax

y=abs(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$, $\mathbb{R}^{m\times n}$,$\mathbb{C}^{m\times n}$, $\mathbb{R}^{m\times n\times p}$, $\mathbb{C}^{m\times n\times p}$	$]-\infty,+\infty[$	$\surd$

Description

This function calculates the absolute value of a real or complex number, vector or matrix.

For $x \in \mathbb{R}$: $y = \begin{cases} x & for\, x \geq 0 \\ -x & for\, x < 0 \end{cases}$

For $\mathbb{C} \ni x := a + i\,b \wedge a, b \in \mathbb{R}$: $y = \sqrt{a^2 + b^2}$

For x being a vector or a matrix the two equations above are applied to the components of x.

Examples

`y=abs(-3)` returns 3,

`y=abs(-3+4*i)` returns 5.

See also

mag(), norm(), real(), imag(), conj(), phase(), arg(), hypot()

angle()

Phase angle in radians of a complex number. Synonym for "arg".

Syntax

y=angle(x)

See also

abs(), mag(), norm(), real(), imag(), conj(), phase(), arg()

arg()

Phase angle in radians of a complex number.

Syntax

y=arg(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$, $\mathbb{R}^{m\times n}$,$\mathbb{C}^{m\times n}$, $\mathbb{R}^{m\times n\times p}$, $\mathbb{C}^{m\times n\times p}$	$]-\infty,+\infty[$	$\surd$

Description

This function returns the phase angle in degrees of a real or complex number, vector or matrix.

For $x \in \mathbb{R}$: $y = \begin{cases} 0 & for\ x \geq 0 \\ \pi & for\ x < 0 \end{cases}$

For $\mathbb{C} \ni x := a + i\,b \wedge a, b \in \mathbb{R}$:

Definition range	Result
$a > 0,\ b > 0$	$y = \arctan\left(\frac{b}{a}\right)$
$a < 0,\ b > 0$	$y = \arctan\left(\frac{b}{a}\right) + \pi$
$a < 0,\ b < 0$	$y = \arctan\left(\frac{b}{a}\right) - \pi$
$a > 0,\ b < 0$	$y = \arctan\left(\frac{b}{a}\right)$
$a = 0,\ b > 0$	$y = \frac{\pi}{2}$
$a > 0,\ b > 0$	$y = -\frac{\pi}{2}$
$a = 0,\ b = 0$	$y = 0$

In this case the arctan() function returns values in radians. The result y of the phase function is in the range $[-\pi,\ +\pi]$. For x being a vector or a matrix the two equations above are applied to the components of x.

Examples

`y=arg(-3)` returns 3.14,

`y=arg(-3+4*i)` returns 2.21.

See also

abs(), mag(), norm(), real(), imag(), conj(), phase()

conj()

Conjugate of a complex number.

Syntax

y=conj(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$, $\mathbb{R}^{m\times n}$,$\mathbb{C}^{m\times n}$, $\mathbb{R}^{m\times n\times p}$, $\mathbb{C}^{m\times n\times p}$	$]-\infty,+\infty[$	$\surd$

Description

This function returns the conjugate of a real or complex number, vector or matrix.

For $x \in \mathbb{R}$: $y = x$

For $\mathbb{C} \ni x := a + i\,b \wedge a, b \in \mathbb{R}$: $y = a - i\,b$

For x being a vector or a matrix the two equations above are applied to the components of x.

Example

`y=conj(-3+4*i)` returns -3-4*i.

See also

abs(), mag(), norm(), real(), imag(), phase(), arg()

deg2rad()

Converts phase from degrees into radians.

Syntax

y=deg2rad(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function converts a real phase, a complex phase or a phase vector given in degrees into radians.

For $x \in \mathbb{R}$: $y = \frac{\pi}{180}\, x$

For $x \in \mathbb{C}$: $y = \frac{\pi}{180}\, Re\,\{x\}$

For x being a vector the two equations above are applied to the components of x.

Example

`y=deg2rad(45)` returns 0.785.

See also

rad2deg(), phase(), arg()

hypot()

Euclidean distance function.

Syntax

z=hypot(x,y)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$
y	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function calculates the Euclidean distance z between two real or complex numbers or vectors. For two numbers $x, y \in \mathbb{C}$, this is

$$z = \sqrt{|x|^2 + |y|^2}$$

For x, y being vectors (of same size) the equation above is applied componentwise.

Examples

`z=hypot(3,4)` returns 5,

`z=hypot(1+2*i,1-2*i)` returns 3.16.

See also

abs()

imag()

Imaginary value of a complex number.

Syntax

y=imag(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$, $\mathbb{R}^{m\times n}$,$\mathbb{C}^{m\times n}$, $\mathbb{R}^{m\times n\times p}$, $\mathbb{C}^{m\times n\times p}$	$]-\infty,+\infty[$	$\surd$

Description

This function returns the imaginary value of a real or complex number, vector or matrix.

For $x \in \mathbb{R}$: $y = 0$

For $\mathbb{C} \ni x := a + i\,b \wedge a, b \in \mathbb{R}$: $y = b$

For x being a vector or a matrix the two equations above are applied to the components of x.

Example

`y=imag(-3+4*i)` returns 4.

See also

abs(), mag(), norm(), real(), conj(), phase(), arg()

mag()

Magnitude of a complex number.

Syntax

y=mag(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$, $\mathbb{R}^{m\times n}$,$\mathbb{C}^{m\times n}$, $\mathbb{R}^{m\times n\times p}$, $\mathbb{C}^{m\times n\times p}$	$]-\infty,+\infty[$	$\surd$

Description

This function calculates the magnitude (absolute value) of a real or complex number, vector or matrix.

For $x \in \mathbb{R}$: $y = \begin{cases} x & for\, x \geq 0 \\ -x & for\, x < 0 \end{cases}$

For $\mathbb{C} \ni x := a + i\,b \wedge a, b \in \mathbb{R}$: $y = \sqrt{a^2 + b^2}$

For x being a vector or a matrix the two equations above are applied to the components of x.

Examples

`y=mag(-3)` returns 3,

`y=mag(-3+4*i)` returns 5.

See also

abs(), norm(), real(), imag(), conj(), phase(), arg()

norm()

Square of the absolute value of a vector.

Syntax

y=norm(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function returns the square of the absolute value of a real or complex number, vector or matrix.

For $x \in \mathbb{R}$: $y = x^2$

For $\mathbb{C} \ni x := a + i\,b \wedge a, b \in \mathbb{R}$: $y = a^2 + b^2$

For x being a vector or a matrix the two equations above are applied to the components of x.

Example

`y=norm(-3+4*i)` returns 25.

See also

abs(), mag(), real(), imag(), conj(), phase(), arg()

phase()

Phase angle in degrees of a complex number.

Syntax

y=phase(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$, $\mathbb{R}^{m\times n}$,$\mathbb{C}^{m\times n}$, $\mathbb{R}^{m\times n\times p}$, $\mathbb{C}^{m\times n\times p}$	$]-\infty,+\infty[$	$\surd$

Description

This function returns the phase angle in degrees of a real or complex number, vector or matrix.

For $x \in \mathbb{R}$: $y = \begin{cases} 0 & for\, x \geq 0 \\ 180 & for\, x < 0 \end{cases}$

For $\mathbb{C} \ni x := a + i\,b \wedge a, b \in \mathbb{R}$:

Definition range	Result
$a > 0,\ b > 0$	$y = \arctan\left(\frac{b}{a}\right)$
$a < 0,\ b > 0$	$y = \arctan\left(\frac{b}{a}\right) + 180$
$a < 0,\ b < 0$	$y = \arctan\left(\frac{b}{a}\right) - 180$
$a > 0,\ b < 0$	$y = \arctan\left(\frac{b}{a}\right)$
$a = 0,\ b > 0$	$y = 90$
$a > 0,\ b > 0$	$y = -90$
$a = 0,\ b = 0$	$y = 0$

In this case the arctan() function returns values in degrees. The result y of the phase function is in the range $[-180,\ +180]$. For x being a vector or a matrix the two equations above are applied to the components of x.

Examples

`y=phase(-3)` returns 180,

`y=phase(-3+4*i)` returns 127.

See also

abs(), mag(), norm(), real(), imag(), conj(), arg()

polar()

Transform from polar coordinates into complex number.

Syntax

c=polar(a,p)

Arguments

Name	Type	Def. Range	Required
a	$\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$
p	$\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function transforms a point given in polar coordinates (amplitude a and phase p in degrees) in the complex plane into the corresponding complex number:

$$x + i\,y = a\,e^{ip} = a\,\cos p + i\,a\,\sin p$$

For a or p being vectors the equation above is applied to the components of a or p.

Example

`c=polar(3,45)` returns 2.12+j2.12.

See also

abs(), mag(), norm(), real(), imag(), conj(), phase(), arg(), exp(), cos(), sin()

rad2deg()

Converts phase from degrees into radians.

Syntax

y=rad2deg(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function converts a real phase, a complex phase or a phase vector given in radians into degrees.

For $x \in \mathbb{R}$: $y = \dfrac{180}{\pi}\, x$

For $x \in \mathbb{C}$: $y = \dfrac{180}{\pi}\, Re\,\{x\}$

For x being a vector the two equations above are applied to the components of x.

Example

`y=rad2deg(45)` returns 0.785.

See also

deg2rad(), phase(), arg()

real()

Real value of a complex number.

Syntax

y=real(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$, $\mathbb{R}^{m\times n}$,$\mathbb{C}^{m\times n}$,$\mathbb{R}^{m\times n\times p}$, $\mathbb{C}^{m\times n\times p}$	$]-\infty,+\infty[$	$\surd$

Description

This function returns the real value of a real or complex number, vector or matrix.

For $x \in \mathbb{R}$: $y = x$

For $\mathbb{C} \ni x := a + i\,b \wedge a, b \in \mathbb{R}$: $y = a$

For x being a vector or a matrix the two equations above are applied to the components of x.

Example

`y=real(-3+4*i)` returns -3.

See also

abs(), mag(), norm(), imag(), conj(), phase(), arg()

signum()

Signum function.

Syntax

y=signum(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function calculates the sign of a real or complex number or vector.

For $x \in \mathbb{R}$: $y = \begin{cases} 1 & for\ x > 0 \\ 0 & for\ x = 0 \\ -1 & for\ x < 0 \end{cases}$

For $x \in \mathbb{C}$: $y = \begin{cases} \dfrac{x}{|x|} & for\ x \neq 0 \\ 0 & for\ x = 0 \end{cases}$

For x being a vector the two equations above are applied to the components of x.

Examples

`y=signum(-4)` returns -1,

`y=signum(3+4*i)` returns 0.6+j0.8.

See also

abs(), sign()

sign()

Sign function.

Syntax

y=sign(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function calculates the sign of a real or complex number or vector.

For $x \in \mathbb{R}$: $y = \begin{cases} 1 & for\ x >= 0 \\ -1 & for\ x < 0 \end{cases}$

For $x \in \mathbb{C}$: $y = \begin{cases} \frac{x}{|x|} & for\ x \neq 0 \\ 1 & for\ x = 0 \end{cases}$

For x being a vector the two equations above are applied to the components of x.

Examples

`y=sign(-4)` returns -1,
`y=sign(3+4*i)` returns 0.6+j0.8.

See also

abs(), signum()

sqr()

Square of a number.

Syntax

y=sqr(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function calculates the square root of a real or complex number or vector.

$y = x^2$

For x being a vector the two equations above are applied to the components of x.

Examples

`y=sqr(-4)` returns 16,
`y=sqr(3+4*i)` returns -7+j24.

See also

sqrt()

sqrt()

Square root.

Syntax

y=sqrt(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function calculates the square root of a real or complex number or vector.

For $x \in \mathbb{R}$: $y = \begin{cases} \sqrt{x} \quad for\, x \geq 0 \\ i\sqrt{-x}\, for\, x < 0 \end{cases}$

For $x \in \mathbb{C}$: $y = \sqrt{|x|}\, e^{i\frac{\varphi}{2}}$ with $\varphi = \arg(x)$

For x being a vector the two equations above are applied to the components of x.

Examples

`y=sqrt(-4)` returns 0+j2,
`y=sqrt(3+4*i)` returns 2+j1.

See also

sqr()

unwrap()

Unwraps a phase vector in radians.

Syntax

y=unwrap(x)

y=unwrap(x, t)

Arguments

Name	Type	Def. Range	Required	Default
x	$\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$	
t	$\mathbb{R}$	$]-\infty,+\infty[$		π

Description

This function unwraps a phase vector x to avoid phase jumps. If two consecutive values of x differ by more than tolerance t, $\mp 2\pi$(depending on the sign of the difference) is added to the current element of x. The predefined value of the optional parameter t is π.

Examples

`y=unwrap(3.15*linspace(-2,2,5))` returns -6.3, -9.43, -12.6, -15.7, -18.8,

`y=unwrap(2*linspace(-2,2,5),1)` returns -4, -8.28, -12.6, -16.8, -21.1,

`y=unwrap(2*linspace(-2,2,5),3)` returns -4, -2, 0, 2, 4.

See also

abs(), mag(), norm(), real(), imag(), conj(), phase(), arg()

Exponential and Logarithmic Functions

exp()

Exponential function.

Syntax

y=exp(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function calculates the exponential function of a real or complex number or vector.

For $x \in \mathbb{R}$: $y = e^x$

For $\mathbb{C} \ni x := a + i\,b \wedge a, b \in \mathbb{R}$: $y = e^x = e^{a+i\,b} = e^a\ (\cos b + i\ \sin b)$

For x being a vector the two equations above are applied to the components of x.

Examples

`y=exp(-4)` returns 0.0183,
`y=exp(3+4*i)` returns -13.1-j15.2.

See also

limexp(), ln(), log10(), log2(), cos(), sin()

limexp()

Limited exponential function.

Syntax

y=limexp(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function is equivalent to the exponential function $\exp(x)$, as long as $x <= 80$. For larger arguments x, it limits the result to $y = \exp(80) \cdot (1 + x - 80)$. The argument can be a real or complex number or vector.

For $x \in \mathbb{R}$: $y = e^x$ for $x \leq 80$, $y = e^{80} \cdot (1 + x - 80)$ else.

For $\mathbb{C} \ni x := a + i\,b \wedge a, b \in \mathbb{R}$: $y = \text{limexp}(x) = \text{limexp}(a + i\,b) = \text{limexp}(a)\ (\cos b + i\,\sin b)$

For x being a vector the two equations above are applied to the components of x.

Examples

`y=limexp(81)` returns 1.11e+35, whereas `y=exp(81)` returns 1.51e+35, which shows the limiting effect of the limexp() function.

`y=limexp(3+4*i)` returns -13.1-j15.2.

See also

exp(), ln(), log10(), log2(), cos(), sin()

log10()

Decimal logarithm.

Syntax

y=log10(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[\setminus \{0\}$	$\surd$

Description

This function calculates the principal value of the decimal logarithm (base 10) of a real or complex number or vector.

For $x \in \mathbb{R}$: $y = \begin{cases} \dfrac{\ln(x)}{\ln(10)} & for\ x > 0 \\ \dfrac{\ln(-x)}{\ln(10)} + i\,\dfrac{\pi}{\ln(10)} & for\ x < 0 \end{cases}$

For $x \in \mathbb{C}$: $y = \dfrac{\ln(|x|)}{\ln(10)} + i\,\dfrac{\arg(x)}{\ln(10)}$

For x being a vector the two equations above are applied to the components of x.

Examples

`y=log10(-4)` returns 0.602+j1.36,

`y=log10(3+4*i)` returns 0.699+j0.403.

See also

ln(), log2(), exp(), arg()

log2()

Binary logarithm.

Syntax

y=log2(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[\setminus \{0\}$	$\surd$

Description

This function calculates the principal value of the binary logarithm (base 2) of a real or complex number or vector.

For $x \in \mathbb{R}$: $y = \begin{cases} \dfrac{\ln(x)}{\ln(2)} & for\ x > 0 \\ \dfrac{\ln(-x)}{\ln(2)} + i\,\dfrac{\pi}{\ln(2)} & for\ x < 0 \end{cases}$

For $x \in \mathbb{C}$: $y = \dfrac{\ln(|x|)}{\ln(2)} + i\,\dfrac{\arg(x)}{\ln(2)}$

For x being a vector the two equations above are applied to the components of x.

Examples

`y=log2(-4)` returns 2+j4.53,

`y=log2(3+4*i)` returns 2.32+j1.34.

See also

ln(), log10(), exp(), arg()

ln()

Natural logarithm (base e).

Syntax

y=ln(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[\setminus \{0\}$	$\surd$

Description

This function calculates the principal value of the natural logarithm (base e) of a real or complex number or vector.

For $x \in \mathbb{R}$: $y = \begin{cases} \ln(x) & for\ x > 0 \\ \ln(-x) & for\ x < 0 \end{cases}$

For $x \in \mathbb{C}$: $y = \ln(|x|) + i\ \arg(x)$

For x being a vector the two equations above are applied to the components of x.

Examples

`y=ln(-4)` returns 1.39+j3.14,

`y=ln(3+4*i)` returns 1.61+j0.927.

See also

log2(), log10(), exp(), arg()

Trigonometry

cos()

Cosine function.

Syntax

y=cos(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty, +\infty[$	$\surd$

Description

This function calculates the cosine of a real or complex number or vector.

For $x \in \mathbb{R}$: $y = \cos(x)$ with $y \in [-1, 1]$

For $x \in \mathbb{C}$: $y = \frac{1}{2}\left(\exp(i\,x) + \exp(-i\,x)\right)$

For x being a vector the two equations above are applied to the components of x.

Examples

`y=cos(-0.5)` returns 0.878,
`y=cos(3+4*i)` returns -27.0-j3.85.

See also

sin(), tan(), arccos()

cosec()

Cosecant.

Syntax

y=cosec(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[\setminus \{k\pi\},\ k \in \mathbb{Z}$	$\surd$

Description

This function calculates the cosecant of a real or complex number or vector.

$$y = \operatorname{cosec} x = \frac{1}{\sin x}$$

For x being a vector the equation above is applied to the components of x.

Example

`y=cosec(1)` returns 1.19.

See also

sin(), sec()

cot()

Cotangent function.

Syntax

y=cot(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[\setminus \{k\pi\},\ k \in \mathbb{Z}$	$\surd$

Description

This function calculates the cotangent of a real or complex number or vector.

For $x \in \mathbb{R}$: $y = \dfrac{1}{\tan(x)}$ with $y \in [-\infty, +\infty]$

For $x \in \mathbb{C}$: $y = i\left(\dfrac{\exp(i\,x)^2 + 1}{\exp(i\,x)^2 - 1}\right)$

For x being a vector the two equations above are applied to the components of x.

Examples

`y=cot(-0.5)` returns -1.83,
`y=cot(3+4*i)` returns -0.000188-j1.

See also

tan(), sin(), cos(), arctan(), arccot()

sec()

Secant.

Syntax

y=sec(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[\setminus \left\{\left(k+\frac{1}{2}\right)\pi\right\},\ k\in\mathbb{Z}$	$\surd$

Description

This function calculates the secant of a real or complex number or vector.

$$y = \text{sec } x = \frac{1}{\cos\ x}$$

For x being a vector the equation above is applied to the components of x.

Example

`y=sec(0)` returns 1.

See also

cos(), cosec()

sin()

Sine function.

Syntax

y=sin(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty, +\infty[$	$\surd$

Description

This function calculates the sine of a real or complex number or vector.

For $x \in \mathbb{R}$: $y = \sin(x)$ with $y \in [-1, 1]$

For $x \in \mathbb{C}$: $y = \frac{1}{2}\, i\, (\exp(-i\, x) - \exp(i\, x))$

For x being a vector the two equations above are applied to the components of x.

Examples

`y=sin(-0.5)` returns -0.479,
`y=sin(3+4*i)` returns 3.85-j27.

See also

cos(), tan(), arcsin()

tan()

Tangent function.

Syntax

y=tan(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[\setminus \left\{\left(k+\frac{1}{2}\right)\pi\right\},\ k \in \mathbb{Z}$	$\surd$

Description

This function calculates the tangent of a real or complex number or vector.

For $x \in \mathbb{R}$: $y = \tan(x)$ with $y \in [-\infty, +\infty]$

For $x \in \mathbb{C}$: $y = -i\left(\frac{\exp(i\,x)^2 - 1}{\exp(i\,x)^2 + 1}\right)$

For x being a vector the two equations above are applied to the components of x.

Examples

`y=tan(-0.5)` returns -0.546,
`y=tan(3+4*i)` returns -0.000187+j0.999.

See also

cot(), sin(), cos(), arctan(), arccot()

Inverse Trigonometric Functions

arccos()

Arc cosine (also known as "inverse cosine").

Syntax

y=arccos(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$[-1, +1]$	$\surd$

Description

This function calculates principal value of the the arc cosine of a real or complex number or vector.

For $x \in \mathbb{R}$: $y = \arccos(x)$ with $y \in [0, \pi]$

For $x \in \mathbb{C}$: $y = -i \ln\left(x + \sqrt{x^2 - 1}\right)$

For x being a vector the two equations above are applied to the components of x.

Examples

`y=arccos(-1)` returns 3.14,
`y=arccos(3+4*i)` returns 0.937-j2.31.

See also

cos(), arcsin(), arctan(), arccot()

arccosec()

Arc cosecant (also known as "inverse cosecant").

Syntax

y=arccosec(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$\mathbb{C}\setminus\{0\}$	$\surd$

Description

This function calculates the principal value of the the arc cosecant of a real or complex number or vector.

For $x \in \mathbb{R}$: $y = \text{arccosec}\,(x)$ with $y \in \left[-\frac{\pi}{2},\, \frac{\pi}{2}\right]$

For $x \in \mathbb{C}$: $y = -i\,\ln\left[\sqrt{1-\frac{1}{x^2}} + \frac{i}{x}\right]$

For x being a vector the two equations above are applied to the components of x.

Examples

`y=arccosec(-1)` returns -1.57,

`y=arccosec(3+4*i)` returns 0.119-j0.16.

See also

cosec(), arcsec()

arccot()

Arc cotangent.

Syntax

y=arccot(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty, +\infty[$	$\surd$

Description

This function calculates the principal value of the arc cotangent of a real or complex number or vector.

For $x \in \mathbb{R}$: $y =$arccot(x) with $y \in [0, \pi]$

For $x \in \mathbb{C}$: $y = \frac{i}{2} \ln\left(\frac{x-i}{x+i}\right)$

For x being a vector the two equations above are applied to the components of x.

Examples

`y=arccot(-1)` returns 2.36,

`y=arccot(3+4*i)` returns 0.122-j0.159.

See also

cot(), tan(), arccos(), arcsin(), arctan()

arcsec()

Arc secant (also known as “inverse secant”).

Syntax

y=arcsec(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$\mathbb{C}\setminus\{0\}$	$\surd$

Description

This function calculates the principal value of the arc secant of a real or complex number or vector.

For $x \in \mathbb{R}$: $y = \operatorname{arcsec}(x)$ with $y \in [0, \pi]$

For $x \in \mathbb{C}$: $y = \frac{\pi}{2} + i \ln\left[\sqrt{1 - \frac{1}{x^2}} + \frac{i}{x}\right]$

For x being a vector the two equations above are applied to the components of x.

Examples

`y=arcsec(-1)` returns 3.14,

`y=arcsec(3+4*i)` returns 1.45+j0.16.

See also

sec(), arccosec()

arcsin()

Arc sine (also known as "inverse sine").

Syntax

y=arcsin(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$[-1, +1]$	$\surd$

Description

This function calculates the principal value of the arc sine of a real or complex number or vector.

For $x \in \mathbb{R}$: $y = \arcsin(x)$ with $y \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$

For $x \in \mathbb{C}$: $y = -i \ln\left[i\,x + \sqrt{1 - x^2}\right]$

For x being a vector the two equations above are applied to the components of x.

Examples

`y=arcsin(-1)` returns -1.57,

`y=arcsin(3+4*i)` returns 0.634+j2.31.

See also

sin(), arccos(), arctan(), arccot()

arctan()

Arc tangent (also known as “inverse tangent”).

Syntax

z=arctan(x)

z=arctan(y,x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$
y	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	

Description

For the first syntax (z=arctan(x)), this function calculates the principal value of the arc tangent of a real or complex number or vector.

For $x \in \mathbb{R}$: $y = \arctan(x)$ with $y \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$

For $x \in \mathbb{C}$: $y = -\frac{1}{2}\, i\, \ln\left[\frac{2\,i}{x+i} - 1\right]$

For x being a vector the two equations above are applied to the components of x.

If the second syntax (z=arctan(y, x)) finds application, the expression

$z = \pm \arctan(y/x)$

(with the arctan() function defined above) is evaluated. The sign of z is determined by

$$\text{sign}(z)=\begin{cases} + & for\, Re\{x\} > 0 \\ - & for\, Re\{x\} > 0 \end{cases}.$$

Note that for the second syntax the case $x = y = 0$ is not defined.

Examples

`z=arctan(-1)` returns -0.785,

`z=arctan(3+4*i)` returns 1.45+j0.159,
`z=arctan(1,1)` returns 0.785.

See also

tan(), arccos(), arcsin(), arccot()

Hyperbolic Functions

cosh()

Hyperbolic cosine.

Syntax

y=cosh(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function calculates the hyperbolic cosine of a real or complex number or vector.

$y = \frac{1}{2}\left(e^{x} + e^{-x}\right)$

For x being a vector the equation above is applied to the components of x.

Examples

`y=cosh(-1)` returns 1.54,
`y=cosh(3+4*i)` returns -6.58-j7.58.

See also

exp(), sinh(), tanh(), cos()

cosech()

Hyperbolic cosecant.

Syntax

y=cosech(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[\setminus\{0\}$	$\surd$

Description

This function calculates the hyperbolic cosecant of a real or complex number or vector.

$$y = \frac{1}{\sinh\, x}$$

For x being a vector the equation above is applied to the components of x.

Examples

`y=cosech(-1)` returns -0.851,
`y=cosech(3+4*i)` returns -0.0649+j0.0755.

See also

exp(), sinh(), sech(), cosec()

coth()

Hyperbolic cotangent.

Syntax

y=coth(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[\setminus\{0\}$	$\surd$

Description

This function calculates the hyperbolic cotangent of a real or complex number or vector.

$$y = \frac{1}{\tanh\, x} = \frac{e^x + e^{-x}}{e^x - e^{-x}}$$

For x being a vector the equation above is applied to the components of x.

Examples

`y=coth(-1)` returns -1.31,
`y=coth(3+4*i)` returns 0.999-j0.0049.

See also

exp(), cosh(), sinh(), tanh(), tan()

sech()

Hyperbolic secant.

Syntax

y=sech(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function calculates the hyperbolic secant of a real or complex number or vector.

$$y = \frac{1}{\cosh\, x}$$

For x being a vector the equation above is applied to the components of x.

Examples

`y=sech(-1)` returns 0.648,
`y=sech(3+4*i)` returns -0.0653+j0.0752.

See also

exp(), cosh(), cosech(), sec()

sinh()

Hyperbolic sine.

Syntax

y=sinh(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function calculates the hyperbolic sine of a real or complex number or vector.

$y = \frac{1}{2}\left(e^{x} - e^{-x}\right)$

For x being a vector the equation above is applied to the components of x.

Examples

`y=sinh(-1)` returns -1.18,
`y=sinh(3+4*i)` returns -6.55-j7.62.

See also

exp(), cosh(), tanh(), sin()

tanh()

Hyperbolic tangent.

Syntax

y=tanh(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function calculates the hyperbolic tangent of a real or complex number or vector.

$$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

For x being a vector the equation above is applied to the components of x.

Examples

`y=tanh(-1)` returns -0.762,
`y=tanh(3+4*i)` returns 1+j0.00491.

See also

exp(), cosh(), sinh(), coth(), tan()

Inverse Hyperbolic Functions

arcosh()

Hyperbolic area cosine.

Syntax

y=arcosh(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$[1, +\infty[$	$\surd$

Description

This function calculates the hyperbolic area cosine of a real or complex number or vector, which is the inverse function to the "cosh" function.

$$y = \operatorname{arcosh} x = \ln\left(x + \sqrt{x^2 - 1}\right)$$

For x being a vector the equation above is applied to the components of x.

Examples

`y=arcosh(1)` returns 0,

`y=arcosh(3+4*i)` returns 2.31+j0.937.

See also

arsinh(), artanh(), cosh(), arccos(), ln(), sqrt()

arcosech()

Hyperbolic area cosecant.

Syntax

y=arcosech(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$\mathbb{C}\setminus\{0\}$	$\surd$

Description

This function calculates the hyperbolic area cosecant of a real or complex number or vector, which is the inverse function to the "cosech" function.

For $x \in \mathbb{C}\setminus\{0\}$: $y = \ln\left(\sqrt{1+\frac{1}{x^2}}+\frac{1}{x}\right)$

For x being a vector the equation above is applied to the components of x.

Examples

`y=arcosech(1)` returns 0.881,

`y=arcosech(i)` returns -i1.57.

See also

cosech(), arsech(), ln(), sqrt()

arcoth()

Hyperbolic area cotangent.

Syntax

y=arcoth(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,-1[\cup]+1,+\infty[$	$\surd$

Description

This function calculates the hyperbolic area cotangent of a real or complex number or vector, which is the inverse function to the "cotanh" function.

$$y = \operatorname{arcoth} x = \frac{1}{2} \ln\left(\frac{x+1}{x-1}\right)$$

For x being a vector the equation above is applied to the components of x.

Examples

`y=arcoth(2)` returns 0.549,

`y=arcoth(3+4*i)` returns 0.118-j0.161.

See also

arsinh(), arcosh(), tanh(), arctan(), ln(), sqrt()

arsech()

Hyperbolic area secant.

Syntax

y=arsech(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$\mathbb{C}\setminus\{0\}$	$\surd$

Description

This function calculates the hyperbolic area secant of a real or complex number or vector, which is the inverse function to the "sech" function.

For $x \in \mathbb{C}\setminus\{0\}$: $y = \ln\left(\sqrt{\frac{1}{x}-1}\sqrt{\frac{1}{x}+1}+\frac{1}{x}\right)$

For x being a vector the equation above is applied to the components of x.

Examples

`y=arsech(1)` returns 0,

`y=arsech(3+4*i)` returns 0.16-j1.45.

See also

sech(), arcosech(), ln(), sqrt()

arsinh()

Hyperbolic area sine.

Syntax

y=arsinh(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function calculates the hyperbolic area sine of a real or complex number or vector, which is the inverse function to the "sinh" function.

$$y = \operatorname{arsinh} x = \ln\left(x + \sqrt{x^2+1}\right)$$

For x being a vector the equation above is applied to the components of x.

Examples

`y=arsinh(1)` returns 0.881,

`y=arsinh(3+4*i)` returns 2.3+j0.918.

See also

arcosh(), artanh(), sinh(), arcsin(), ln(), sqrt()

artanh()

Hyperbolic area tangent.

Syntax

y=artanh(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-1,+1[$	$\surd$

Description

This function calculates the hyperbolic area tangent of a real or complex number or vector, which is the inverse function to the "tanh" function.

$$y = \operatorname{artanh} x = \frac{1}{2} \ln\left(\frac{1+x}{1-x}\right)$$

For x being a vector the equation above is applied to the components of x.

Examples

`y=artanh(0)` returns 0,

`y=artanh(3+4*i)` returns 0.118+j1.41.

See also

arsinh(), arcosh(), tanh(), arctan(), ln(), sqrt()

Rounding

ceil()

Round to the next higher integer.

Syntax

y=ceil(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function rounds a real number x to the next higher integer value.

If x is a complex number both real part and imaginary part are rounded. For x being a vector the operation above is applied to the components of x.

Examples

`y=ceil(-3.5)` returns -3,

`y=ceil(3.2+4.7*i)` returns 4+j5.

See also

floor(), fix(), round()

fix()

Truncate decimal places from real number.

Syntax

y=fix(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function truncates the decimal places from a real number x and returns an integer.

If x is a complex number both real part and imaginary part are rounded. For x being a vector the operation above is applied to the components of x.

Examples

`y=fix(-3.5)` returns -3,

`y=fix(3.2+4.7*i)` returns 3+j4.

See also

ceil(), floor(), round()

floor()

Round to the next lower integer.

Syntax

y=floor(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function rounds a real number x to the next lower integer value.

If x is a complex number both real part and imaginary part are rounded. For x being a vector the operation above is applied to the components of x.

Examples

`y=floor(-3.5)` returns -4,

`y=floor(3.2+4.7*i)` returns 3+j4.

See also

ceil(), fix(), round()

round()

Round to nearest integer.

Syntax

y=round(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function rounds a real number x to its nearest integer value.

If x is a complex number both real part and imaginary part are rounded. For x being a vector the operation above is applied to the components of x.

Examples

`y=round(-3.5)` returns -4,

`y=round(3.2+4.7*i)` returns 3+j5.

See also

ceil(), floor(), fix()

besseli0()

Modified Bessel function of order zero.

Syntax

i0=besseli0(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function evaluates the modified Bessel function of order zero of a real or complex number or vector.

$$i0\left(x\right)=J_0\left(i\,x\right)=\sum_{k=0}^{\infty}\frac{\left(\frac{x}{2}\right)^{2k}}{k!\,\Gamma\left(k+1\right)},$$

where $J_0\left(x\right)$is the Bessel function of order zero and $\Gamma\left(x\right)$denotes the gamma function.

For x being a vector the equation above is applied to the components of x.

Example

`y=besseli0(1)` returns 1.266.

See also

besselj(), bessely()

besselj()

Bessel function of n-th order.

Syntax

jn=besselj(n,x)

Arguments

Name	Type	Def. Range	Required
n	$\mathbb{N}$	$[0,+\infty[$	$\surd$
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function evaluates the Bessel function of n-th order of a real or complex number or vector.

$$J_n(x) = \sum_{k=0}^{\infty} \frac{(-1)^k \left(\frac{x}{2}\right)^{n+2k}}{k!\,\Gamma(n+k+1)},$$

where $\Gamma(x)$denotes the gamma function.

For x being a vector the equation above is applied to the components of x.

Example

`y=besselj(1,1)` returns 0,44.

See also

besseli0(), bessely()

bessely()

Bessel function of second kind and n-th order.

Syntax

yn=bessely(n,x)

Arguments

Name	Type	Def. Range	Required
n	$\mathbb{N}$	$[0,+\infty[$	$\surd$
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function evaluates the Bessel function of second kind and n-th order of a real or complex number or vector.

$$Y_n(x) = \lim_{m \to n} \frac{J_m(x)\,\cos m\pi - J_{-m}(x)}{\sin m\pi},$$

where $J_m(x)$denotes the Bessel function of first kind and n-th order.

For x being a vector the equation above is applied to the components of x.

Example

`y=bessely(1,1)` returns -0.781.

See also

besseli0(), besselj()

erf()

Error function.

Syntax

y=erf(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function evaluates the error function of a real or complex number or vector. For $x \in \mathbb{R}$,

$$y = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

If x is a complex number both real part and imaginary part are subjected to the equation above. For x being a vector the equation is applied to the components of x.

Example

`y=erf(0.5)` returns 0.520.

See also

erfc(), erfinv(), erfcinv(), exp()

erfc()

Complementary error function.

Syntax

y=erfc(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function evaluates the complementary error function of a real or complex number or vector. For $x \in \mathbb{R}$,

$$y = 1 - \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

If x is a complex number both real part and imaginary part are subjected to the equation above. For x being a vector the equation is applied to the components of x.

Example

`y=erfc(0.5)` returns 0.480.

See also

erf(), erfinv(), erfcinv(), exp()

erfinv()

Inverse error function.

Syntax

y=erfinv(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-1,+1[$	$\surd$

Description

This function evaluates the inverse of the error function of a real or complex number or vector. For $-1 < x < 1$,

$y = \mathrm{erf}^{-1}(x)$

If x is a complex number both real part and imaginary part are subjected to the equation above. For x being a vector the equation is applied to the components of x.

Example

`y=erfinv(0.8)` returns 0.906.

See also

erf(), erfc(), erfcinv(), exp()

erfcinv()

Inverse complementary error function.

Syntax

y=erfcinv(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]0,+2[$	$\surd$

Description

This function evaluates the inverse of the complementary error function of a real or complex number or vector. For $0 < x < 2$,

$y = \mathrm{erfc}^{-1}(x)$

If x is a complex number both real part and imaginary part are subjected to the equation above. For x being a vector the equation is applied to the components of x.

Example

`y=erfcinv(0.5)` returns 0.477.

See also

erf(), erfc(), erfinv(), exp()

sinc()

Sinc function.

Syntax

y=sinc(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function evaluates the sinc function of a real or complex number or vector.

$$y = \begin{cases} \dfrac{\sin x}{x} & for\ x \neq 0 \\ 1 & for\ x = 0 \end{cases}$$

For x being a vector the equation above is applied to the components of x.

Examples

`y=sinc(-3)` returns 0.047,

`y=sinc(3+4*i)` returns -3.86-j3.86.

See also

sin()

step()

Step function.

Syntax

y=step(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function calculates the step function of a real or complex number or vector. For $x \in \mathbb{R}$,

$$y = \begin{cases} 0 & for\, x < 0 \\ 0.5 & for\, x = 0 \\ 1 & for\, x > 0 \end{cases}$$

If x is a complex number both real part and imaginary part are subjected to the equation above. For x being a vector the equation is applied to the components of x.

Example

`y=step(0.5)` returns 1.

See also

15.4.3 Data Analysis

Basic Statistics

avg()

Average of vector elements.

Syntax

y=avg(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$, Range $xs : xe$	$]-\infty, +\infty[$	$\surd$

Description

This function returns the sum of the elements of a real or complex vector or range.

For $x \in \mathbb{C}^n$: $y = \frac{1}{n} \sum_{i=1}^{n} x_i$, $1 \leq i \leq n$ (for vectors) or $xs \leq i \leq xe$ (for ranges)

For x being a real or complex number, x itself is returned.

Example

`y=avg(linspace(1,3,10))` returns 2.

See also

sum(), max(), min()

cumavg()

Cumulative average of vector elements.

Syntax

y=cumavg(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function returns the cumulative average of the elements of a real or complex vector.

For $x \in \mathbb{C}^n$: $y_k = \frac{1}{k}\sum_{i=1}^{k} x_i$, $1 \leq k \leq n$

For x being a real or complex number, x itself is returned.

Example

`y=cumavg(linspace(1,3,3))` returns 1, 1.5, 2.

See also

cumsum(), cumprod(), avg(), sum(), prod(), max(), min()

max()

Maximum value.

Syntax

y=max(x)

y=max(a,b)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$, Range $xs:xe$	$]-\infty,+\infty[$	$\surd$
a	$\mathbb{R}$, $\mathbb{C}$	$]-\infty,+\infty[$	$\surd$
b	$\mathbb{R}$, $\mathbb{C}$	$]-\infty,+\infty[$	$\surd$

Description

For the first syntax (y=max(x)), this function returns the maximum value of a real or complex vector or range.

For $x \in \mathbb{R}^n$: $y = \max(x_i),\ 1 \le i \le n$ (for vectors) or $xs \le i \le xe$ (for ranges)

For $x \in \mathbb{C}^n$: $y = \max(\pm|x_i|),\ 1 \le i \le n$ (for vectors) or $xs \le i \le xe$ (for ranges),

with sign $\begin{cases} + & for\ |\arg(x_i)| \le \frac{\pi}{2} \\ - & else \end{cases}$

For x being a real or complex number: that is the case $n = 1$.

The second syntax (y=max(a,b)) finds application, if two (generally complex) numbers a and b need to be compared. In principle, the maximum of the absolute values is selected, but it must be considered whether a and b are located in the right or left complex half plane. If the latter is the case, the negative absolute value of a and b needs to be regarded (for example, which is the case for negative real numbers), otherwise the positive absolute value is taken:

$y = \max(\pm|a|, \pm|b|)$,

with $|a|$ sign $\begin{cases} + & for\ |\arg(a)| \le \frac{\pi}{2} \\ - & else \end{cases}$ and $|b|$ sign $\begin{cases} + & for\ |\arg(b)| \le \frac{\pi}{2} \\ - & else \end{cases}$

Example

`y=max(linspace(1,3,10))` returns 3.

`y=max(1,3)` returns 3.

`y=max(1,1+i)` returns 1+j1.

`y=max(1,-1+i)` returns 1.

See also

min(), abs()

min()

Minimum value.

Syntax

y=min(x)

y=min(a,b)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$, Range $xs : xe$	$]-\infty, +\infty[$	$\surd$
a	$\mathbb{R}$, $\mathbb{C}$	$]-\infty, +\infty[$	$\surd$
b	$\mathbb{R}$, $\mathbb{C}$	$]-\infty, +\infty[$	$\surd$

Description

For the first syntax (y=min(x)), this function returns the minimum value of a real or complex vector or range.

For $x \in \mathbb{R}^n$: $y = \min(x_i),\ 1 \le i \le n$ (for vectors) or $xs \le i \le xe$ (for ranges)

For $x \in \mathbb{C}^n$: $y = \min(\pm|x_i|),\ 1 \le i \le n$ (for vectors) or $xs \le i \le xe$ (for ranges),

with sign $\begin{cases} + & for\ |\arg(x_i)| \le \frac{\pi}{2} \\ - & else \end{cases}$

For x being a real or complex number: that is the case $n = 1$.

The second syntax (y=min(a,b)) finds application, if two (generally complex) numbers a and b need to be compared. In principle, the maximum of the absolute values is selected, but it must be considered whether a and b are located in the right or left complex half plane. If the latter is the case, the negative absolute value of a and b needs to be regarded (for example, which is the case for negative real numbers), otherwise the positive absolute value is taken:

$y = \min(\pm|a|, \pm|b|)$,

with $|a|$ sign $\begin{cases} + & for\ |\arg(a)| \le \frac{\pi}{2} \\ - & else \end{cases}$ and $|b|$ sign $\begin{cases} + & for\ |\arg(b)| \le \frac{\pi}{2} \\ - & else \end{cases}$

Example

`y=min(linspace(1,3,10))` returns 1.

`y=min(1,3)` returns 1.

`y=min(1,1+i)` returns 1.

`y=min(1,-1+i)` returns -1+j1.

See also

max(), abs()

rms()

Root Mean Square of vector elements.

Syntax

y=rms(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function returns the rms (root mean square) value of the elements of a real or complex vector. By application of the trapezoidal integration rule,

$$\text{for } x \in \mathbb{C}^n\text{: } y = \sqrt{\frac{1}{n}\sum_{i=1}^{n} a_i\, x_i\, x_i^*},\ 1 \le i \le n,\ a_i = \begin{cases} 1 & \text{for } 2 \le i \le n-1 \\ \frac{1}{2} & \text{for } i = 1 \text{ or } i = n \end{cases}$$

For x being a real or complex number, $|x|$ itself is returned.

Example

`y=rms(linspace(1,2,8))` returns 1.43.

See also

variance(), stddev(), avg()

runavg()

Running average of vector elements.

Syntax

y=runavg(x,m)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$
m	$\mathbb{N}$	$[1,+\infty[$	$\surd$

Description

This function returns the running average over m elements of a real or complex vector.

For $x \in \mathbb{C}^n$: $y_k = \frac{1}{m} \sum_{i=k}^{k+m-1} x_i$, $1 \leq k \leq n$

For x being a real or complex number, x itself is returned.

Example

`y=runavg(linspace(1,3,6),2)` returns 1.2, 1.6, 2, 2.4, 2.8.

See also

cumavg(), cumsum(), avg(), sum()

stddev()

Standard deviation of vector elements.

Syntax

y=stddev(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function returns the stddev of the elements of a real or complex vector x.

For $x \in \mathbb{C}^n$: $y = \sqrt{\text{variance}(x)}$

For x being a real or complex number, 0 is returned.

Example

`y=stddev(linspace(1,3,10))` returns 0.673.

See also

stddev(), avg(), max(), min()

variance()

Variance of vector elements.

Syntax

y=variance(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function returns the variance of the elements of a real or complex vector.

For $x \in \mathbb{C}^n$: $y = \dfrac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2$, where $\overline{x}$ denotes mean (average) value of x.

For x being a real or complex number, 0 is returned.

Example

`y=variance(linspace(1,3,10))` returns 0.453.

See also

stddev(), avg(), max(), min()

random()

Random number between 0.0 and 1.0.

Syntax

y=random()

Arguments

None.

Description

This function returns a pseudo-random real number between 0.0 (including) and 1.0 (excluding). The starting point of the random number generator can be set by srandom().

Example

```
y=random()
```

See also

srandom()

srandom()

Set seed for a new series of pseudo-random numbers.

Syntax

y=srandom(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$	$]-\infty,+\infty[$	$\surd$

Description

This function establishes x as the seed for a new series of pseudo-random numbers. Please note that only integer values for x are considered, so for example $x = 1.1$ will give the same seed as $x = 1$.

Example

```
y=srandom(100)
```

See also

random()

Basic Operation

cumprod()

Cumulative product of vector elements.

Syntax

y=cumprod(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function returns the cumulative product of the elements of a real or complex vector.

For $x \in \mathbb{C}^n$: $y_k = \prod_{i=1}^{k} x_i$, $1 \leq k \leq n$

For x being a real or complex number, x itself is returned.

Example

`y=cumprod(linspace(1,3,3))` returns 1, 2, 6.

See also

cumsum(), cumavg(), prod(), sum(), avg(), max(), min()

cumsum()

Cumulative sum of vector elements.

Syntax

y=cumsum(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function returns the cumulative sum of the elements of a real or complex vector.

For $x \in \mathbb{C}^n$: $y_k = \sum_{i=1}^{k} x_i$, $1 \leq k \leq n$

For x being a real or complex number, x itself is returned.

Example

`y=cumsum(linspace(1,3,3))` returns 1, 3, 6.

See also

cumprod(), cumavg(), sum(), prod(), avg(), max(), min()

interpolate()

Equidistant spline interpolation of data vector.

Syntax

z=interpolate(y,t,m)

z=interpolate(y,t)

Arguments

Name	Type	Def. Range	Required	Default
y	$\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$	
t	$\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$	
m	$\mathbb{N}$	$[3,+\infty[$		64

Description

This function uses spline interpolation to interpolate between the points of a vector *y(t)*. If the number of samples n is not specified, a default value of $n = 64$ is assumed.

Example

```
z=interpolate(linspace(0,2,3)*linspace(0,2,3),linspace(0,2,3))
```

returns a smooth parabolic curve:

Use the Cartesian diagram to display it.

See also

sum(), prod()

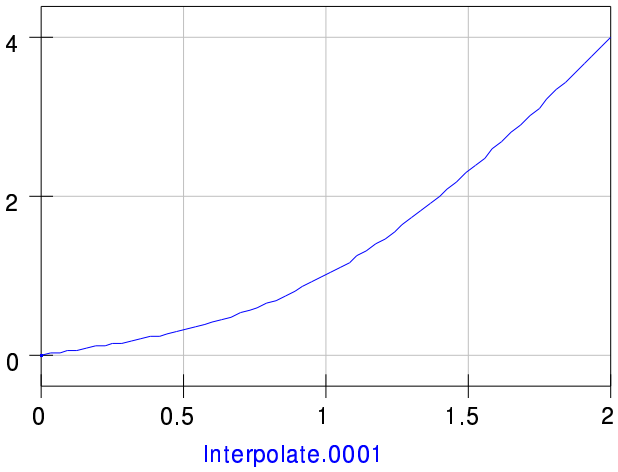


Figure 15.4: Interpolated curve

prod()

Product of vector elements.

Syntax

y=prod(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function returns the product of the elements of a real or complex vector.

For $x \in \mathbb{C}^n$: $y = \prod_{i=1}^{n} x_i$

For x being a real or complex number, x itself is returned.

Example

`y=prod(linspace(1,3,10))` returns 583.

See also

sum(), avg(), max(), min()

sum()

Sum of vector elements.

Syntax

y=sum(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function returns the sum of the elements of a real or complex vector.

For $x \in \mathbb{C}^n$: $y = \sum_{i=1}^{n} x_i$

For x being a real or complex number, x itself is returned.

Example

`y=sum(linspace(1,3,10))` returns 20.

See also

prod(), avg(), max(), min()

xvalue()

Returns x-value which is associated with the y-value nearest to a specified y-value in a given vector.

Syntax

x=xvalue(f,yval)

Arguments

Name	Type	Def. Range	Required
f	$\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$
yval	$\mathbb{R}$, $\mathbb{C}$	$]-\infty,+\infty[$	$\surd$

Description

This function returns the x-value which is associated with the y-value nearest to *yval* in the given vector f; therefore the vector f must have a single data dependency.

Example

```
x=xvalue(f,1).
```

See also

yvalue(), interpolate()

yvalue()

Returns y-value of a given vector which is located nearest to the specified x-value.

Syntax

y=yvalue(f,xval)

Arguments

Name	Type	Def. Range	Required
f	$\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$
xval	$\mathbb{R}$, $\mathbb{C}$	$]-\infty,+\infty[$	$\surd$

Description

This function returns the y-value of the given vector f which is located nearest to the x-value *xval*; therefore the vector f must have a single data dependency.

Example

```
y=yvalue(f,1).
```

See also

xvalue(), interpolate()

Differentiation and Integration

ddx()

Differentiate mathematical expression with respect to a given variable.

Syntax

y=ddx(f(x),x)

Arguments

Name	Type	Def. Range	Required	Default
f(x)			√	
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^m$, $\mathbb{C}^m$	$]-\infty,+\infty[$	√	

Description

This function executes a symbolic differentiation on a function *f(x)* with respect to a variable x. The result is evaluated at the contents x_0 of x.

$$y = \left.\frac{df}{dx}\right|_{x_0}$$

If x is a vector, the differential quotient is evaluated for all components of x, giving a result vector y.

Example

Create a vector x by setting `x=linspace(0,2,3)`, thus $x = [0, 1, 2]^T$. Entering

`y=ddx(sin(x),x` returns 1, 0.54, -0.416.

Why? $\frac{df}{dx} = \frac{d\sin(x)}{dx} = \cos(x)$, and $\cos(x)$ evaluated at $x = [0, 1, 2]^T$ gives the result above.

See also

diff()

diff()

Differentiate vector with respect to another vector.

Syntax

z=diff(y,x,n)

Arguments

Name	Type	Def. Range	Required	Default
y	$\mathbb{R}^k$, $\mathbb{C}^k$	$]-\infty,+\infty[$	$\surd$	
x	$\mathbb{R}^m$, $\mathbb{C}^m$	$]-\infty,+\infty[$	$\surd$	
n	$\mathbb{N}$			1

Description

This function numerically differentiates a vector y with respect to a vector x. If the optional integer parameter n is given, the n-th derivative is calculated. Differentiation is executed for N=min*(k,m)* elements. For n=1,

$$\frac{\Delta y_i}{\Delta x_i} = \begin{cases} \frac{1}{2}\left(\frac{y_i - y_{i-1}}{x_i - x_{i-1}} + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}\right) & \text{for } N-1 > i > 0 \\ \frac{y_{i+1} - y_i}{x_{i+1} - x_i} & \text{for } i = 0 \\ \frac{y_i - y_{i-1}}{x_i - x_{i-1}} & \text{for } i = N-1 \end{cases}$$

If *n>1*, the result of the differentiation above is assigned to y and the aforementioned differentiation step is repeated until the number of those steps is equal to n.

Example

`z=diff(linspace(1,3,3),linspace(2,3,3))` returns 2, 2, 2.

See also

integrate(), sum(), max(), min()

integrate()

Integrate vector.

Syntax

z=integrate(y,h)

Arguments

Name	Type	Def. Range	Required
y	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$
h	$\mathbb{R}$, $\mathbb{C}$	$]-\infty,+\infty[$	$\surd$

Description

This function numerically integrates a vector x with respect to a differential h. The integration method is according to the trapezoidal rule:

$$\int f\left(t\right)dt \approx h\left(\frac{y_0}{2}+y_1+y_2+\ldots+y_{n-1}+\frac{y_n}{2}\right)$$

Example

Calculate an approximation of the integral $\int_1^3 t\,dt$ using 101 points:

`z=integrate(linspace(1,3,101))` returns 4.

See also

diff(), sum(), max(), min()

Signal Processing

dft()

Discrete Fourier Transform.

Syntax

y=dft(v)

Arguments

Name	Type	Def. Range	Required
v	$\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function computes the Discrete Fourier Transform (DFT) of a vector v. The advantage of this function compared to fft() is that the number n of components of v is arbitrary, while for the latter n must be a power of 2. The drawbacks are that dft() is slower and less accurate than fft().

Example

This calculates the spectrum y of a DC signal:

`y=dft(linspace(1,1,7))` returns

y
1
-1.59e-17+j1.59e-17
⋮
2.22e-16-j1.11e-16

Please note that in this example 7 points are used for the time vector v. Since 7 is not a power of 2, the same expression used together with the fft() function would lead to wrong results. Note also the rounding errors where "0" would be the correct value.

See also

idft(), fft(), ifft(), Freq2Time(), Time2Freq()

fft()

Fast Fourier Transform.

Syntax

y=fft(v)

Arguments

Name	Type	Def. Range	Required
v	$\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function computes the Fast Fourier Transform (FFT) of a vector v. The number n of components of v must be a power of 2.

Example

This calculates the spectrum y of a DC signal:

`y=fft(linspace(1,1,8))` returns

y
1
0
⋮
0

See also

ifft(), dft(), idft(), Freq2Time(), Time2Freq(), fftshift()

idft()

Inverse Discrete Fourier Transform.

Syntax

y=idft(v)

Arguments

Name	Type	Def. Range	Required
v	$\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function computes the Inverse Discrete Fourier Transform (IDFT) of a vector v. The advantage of this function compared to ifft() is that the number n of components of v is arbitrary, while for the latter n must be a power of 2. The drawbacks are that idft() is slower and less accurate than ifft().

Example

This calculates the time function y belonging to a white spectrum:

`y=idft(linspace(1,1,7))` returns

y
7
-1.11e-16-j1.11e-16
⋮
1.55e-15+j7.77e-16

Please note that in this example 7 points are used for the spectrum vector v. Since 7 is not a power of 2, the same expression used together with the ifft() function would lead to wrong results. Note also the rounding errors where “0” would be the correct value.

See also

dft(), ifft(), fft(), Freq2Time(), Time2Freq(), fftshift()

ifft()

Inverse Fast Fourier Transform.

Syntax

y=ifft(v)

Arguments

Name	Type	Def. Range	Required
v	$\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function computes the Inverse Fast Fourier Transform (IFFT) of a vector v. The number n of components of v must be a power of 2.

Example

This calculates the time function y belonging to a white spectrum:

`y=ifft(linspace(1,1,8))` returns

y
8
0
⋮
0

See also

fft(), dft(), idft(), Freq2Time(), Time2Freq(), fftshift()

fftshift()

Move the frequency 0 to the center of the FFT vector.

Syntax

y=fftshift(v)

Arguments

Name	Type	Def. Range	Required
v	$\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty, +\infty[$	$\surd$

Description

This function shuffles the FFT values of vector v in order to move the frequency 0 to the center of the vector. Below of it the components with negative frequencies are located, above those with positive frequencies. Herewith the "classical" look of a spectrum as gained by a spectrum analyzer is obtained.

Example

Suppose x to be the result of a FFT of 8 elements, e.g.

x
1
2
⋮
8

The result of the FFT is sorted in such a way that the component with frequency zero is the first element (1) of the vector. The components with positive frequency follow (2,3,4). After that, the components with negative frequency (5,6,7,8) are arranged, starting from the most negative value. This pattern can be exemplarily generated in Qucs by writing `x=linspace(1,8,8)`. Then

`y=fftshift(x)` returns

y
5
6
7
8
1
2
3
4

As you can see, the component with frequency 0 (element 1) is moved to the middle of the spectrum vector. Beneath of it the components with negative frequencies appear (5,6,7,8), above those with positive frequencies (2,3,4).

See also

fft(), ifft(), dft(), idft()

Time2Freq()

Interpreted Discrete Fourier Transform.

Syntax

y=Time2Freq(v,t)

Arguments

Name	Type	Def. Range	Required
v	$\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$
t	$\mathbb{R}^k$, $\mathbb{C}^k$	$]-\infty,+\infty[$	$\surd$

Description

This function computes the Discrete Fourier Transform (DFT) of a vector v with respect to a time vector t.

Example

This calculates the spectrum y(f) of a DC signal:

`y=Time2Freq(linspace(1,1,7),linspace(0,1,2))` returns

Frequency	y
0	1
0.167	-1.59e-17+j1.59e-17
⋮	⋮
1	2.22e-16-j1.11e-16

Please note that in this example 7 points are used for the time vector v. Note also the rounding errors at t>0, where "0" would be the correct value.

See also

idft(), fft(), ifft(), Freq2Time()

Freq2Time()

Interpreted Inverse Discrete Fourier Transform.

Syntax

y=Freq2Time(v,f)

Arguments

Name	Type	Def. Range	Required
v	$\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$
f	$\mathbb{R}^k$, $\mathbb{C}^k$	$]-\infty,+\infty[$	$\surd$

Description

This function computes the Inverse Discrete Fourier Transform (IDFT) of a vector v with respect to a frequency vector f.

Example

This calculates the time function y(t) belonging to a white spectrum:

`y=Freq2Time(linspace(1,1,7),linspace(0,1,2))` returns

Frequency	y
0	7
0.167	-1.11e-16-j1.11e-16
⋮	⋮
1	1.55e-15+j7.77e-16

Please note that in this example 7 points are used for the spectrum vector v. Note also the rounding errors at t>0, where "0" would be the correct value.

See also

dft(), ifft(), fft(), Time2Freq()

kbd()

Kaiser-Bessel derived window.

Syntax

y=kbd(a,n)

y=kbd(a)

Arguments

Name	Type	Def. Range	Required	Default
a	$\mathbb{R}$	$]-\infty,+\infty[$	$\surd$	
n	$\mathbb{N}$	$[1,+\infty[$		64

Description

This function generates a Kaiser-Bessel window according to

$$y_k \quad = \sqrt{\frac{\sum_{i=0}^{k} I_0\left(\pi\, a \sqrt{1-\left(\frac{4\,i}{n}-1\right)}\right)}{\sum_{i=0}^{\frac{n}{2}} I_0\left(\pi\, a \sqrt{1-\left(\frac{4\,i}{n}-1\right)}\right)}},$$

$y_{n-k-1} = y_k$

for $0 \leq k < \frac{n}{2}$

If the parameter n is not specified, n=64 is assumed.

Example

`y=kbd(0.1,4)` returns .

See also

dft(), ifft(), fft()

15.5 Electronics Functions

15.5.1 Unit Conversion

dB()

dB value.

Syntax

y=dB(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function returns the dB value of a real or complex number or vector.

$y = 20 \log |x|$

For x being a vector the equation above is applied to the components of x.

Example

`y=db(10)` returns 20.

See also

log10()

dbm()

Convert voltage to power in dBm.

Syntax

y=dBm(u,Z0)

y=dBm(u)

Arguments

Name	Type	Def. Range	Required	Default
u	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$	
Z0	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$		50

Description

This function returns the corresponding dBm power of a real or complex voltage or vector u. The impedance *Z0* referred to is either specified or 50Ω.

$$y = 10 \log \frac{|u|^2}{Z_0\, 0.001W}$$

For u being a vector the equation above is applied to the components of u.

Please note that u is considered as a rms value, not as an amplitude.

Example

`y=dbm(1)` returns 13.

See also

dbm2w(), w2dbm(), log10()

dbm2w()

Convert power in dBm to power in Watts.

Syntax

y=dBm2w(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function converts the real or complex power or power vector, given in dBm, to the corresponding power in Watts.

$y = 0.001\,10^{\frac{x}{10}}$

For x being a vector the equation above is applied to the components of x.

Example

`y=dbm2w(10)` returns 0.01.

See also

dbm(), w2dbm()

w2dbm()

Convert power in Watts to power in dBm.

Syntax

y=w2dBm(x)

Arguments

Name	Type	Def. Range	Required
x	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function converts the real or complex power or power vector, given in Watts, to the corresponding power in dBm.

$$y = 10 \log \frac{x}{0.001W}$$

For x being a vector the equation above is applied to the components of x.

Example

`y=w2dbm(1)` returns 30.

See also

dbm(), dbm2w(), log10()

15.5.2 Reflection Coefficients and VSWR

rtoswr()

Converts reflection coefficient to voltage standing wave ratio (VSWR).

Syntax

s=rtoswr(r)

Arguments

Name	Type	Def. Range	Required
r	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$\|r\| \leq 1$	$\surd$

Description

For a real or complex reflection coefficient r, this function calculates the corresponding voltage standing wave ratio (VSWR) s according to

$$s = \frac{1 + |r|}{1 - |r|}$$

VSWR is a real number and if usually given in the notation "s : 1".

For r being a vector the equation above is applied to the components of r.

Examples

`s=rtoswr(0)` returns 1.

`s=rtoswr(0.1+0.2*i)` returns 1.58.

See also

ytor(), ztor(), rtoy(), rtoz()

rtoy()

Converts reflection coefficient to admittance.

Syntax

y=rtoy(r)

y=rtoy(r, Z0)

Arguments

Name	Type	Def. Range	Required	Default
r	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$\|r\| \leq 1$	$\surd$	
Z0	$\mathbb{R}$, $\mathbb{C}$	$]-\infty, +\infty[$		50

Description

For a real or complex reflection coefficient r, this function calculates the corresponding admittance y according to

$$y = \frac{1}{Z_0}\,\frac{1-r}{1+r}$$

If the reference impedance *Z0* is not provided, the function assumes $Z0 = 50\Omega$.

For r being a vector the equation above is applied to the components of r.

Example

`y=rtoy(0.333)` returns 0.01.

See also

ytor(), ztor(), rtoswr()

rtoz()

Converts reflection coefficient to impedance.

Syntax

z=rtoz(r)

z=rtoz(r, Z0)

Arguments

Name	Type	Def. Range	Required	Default
r	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$\|r\| \leq 1$	$\surd$	
Z0	$\mathbb{R}$, $\mathbb{C}$	$]-\infty, +\infty[$		50

Description

For a real or complex reflection coefficient r, this function calculates the corresponding impedance Z according to

$$Z = Z_0 \frac{1-r}{1+r}$$

If the reference impedance *Z0* is not provided, the function assumes $Z0 = 50\Omega$.

For r being a vector the equation above is applied to the components of r.

Example

`z=rtoz(0.333)` returns 99.9.

See also

ztor(), ytor(), rtoswr()

ytor()

Converts admittance to reflection coefficient.

Syntax

r=ytor(Y)

r=ytor(Y, Z0)

Arguments

Name	Type	Def. Range	Required	Default
Y	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$	
Z0	$\mathbb{R}$, $\mathbb{C}$	$]-\infty,+\infty[$		50

Description

For a real or complex admittance y, this function calculates the corresponding reflection coefficient according to

$$r = \frac{1 - Y\,Z_0}{1 + Y\,Z_0}$$

For Y being a vector the equation above is applied to the components of Y.

If the reference impedance *Z0* is not provided, the function assumes $Z0 = 50\Omega$.

Often a dB measure is given for the reflection coefficient, the so called "return loss":

$RL = -20\,\log|r|$ [dB]

Example

`r=ytor(0.01)` returns 0.333.

See also

rtoy(), rtoz(), rtoswr(), log10(), dB()

ztor()

Converts impedance to reflection coefficient.

Syntax

r=ztor(Z)

r=ztor(Z, Z0)

Arguments

Name	Type	Def. Range	Required	Default
Z	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$	
Z0	$\mathbb{R}$, $\mathbb{C}$	$]-\infty,+\infty[$		50

Description

For a real or complex impedance Z, this function calculates the corresponding reflection coefficient according to

$$r = \frac{Z - Z_0}{Z + Z_0}$$

For Z being a vector the equation above is applied to the components of Z.

If the reference impedance *Z0* is not provided, the function assumes $Z0 = 50\Omega$.

Often a dB measure is given for the reflection coefficient, the so called “return loss”:

$RL = -20 \log |r|$ [dB]

Example

`r=ztor(100)` returns 0.333.

See also

rtoz(), rtoy(), rtoswr(), log10(), dB()

15.5.3 N-Port Matrix Conversions

stos()

Converts S-parameter matrix to S-parameter matrix with different reference impedance(s).

Syntax

y=stos(S, Zref)

y=stos(S, Zref, Z0)

Arguments

Name	Type	Def. Range	Required	Default
S	$\mathbb{R}^{n\times n}$, $\mathbb{C}^{n\times n}$	$\lvert S_{ij}\rvert \in]-\infty,+\infty[,\ 1\le i,j\le n$ $\lvert S_{ii}\rvert \le 1,\ 1\le i\le n$	√	
Zref	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	√	
Z0	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$		50

Description

This function converts a real or complex scattering parameter matrix S into a scattering matrix Y. S has a reference impedance *Zref*, whereas the created scattering matrix Y has a reference impedance *Z0*.

If the reference impedance *Z0* is not provided, the function assumes $Z0 = 50\Omega$.

Both *Zref* and *Z0* can be real or complex numbers or vectors; in the latter case the function operates on the elements of *Zref* and *Z0*.

Example

Conversion of 50Ω terminated S-parameters to 100Ω terminated S-parameters:

`S2=stos(eye(2)*0.1,50,100)` returns

-0.241	0
0	-0.241

.

See also

twoport(), stoy(), stoz()

stoy()

Converts S-parameter matrix to Y-parameter matrix.

Syntax

Y=stoy(S)

Y=stoy(S, Zref)

Arguments

Name	Type	Def. Range	Required	Default
S	$\mathbb{R}^{n\times n}$, $\mathbb{C}^{n\times n}$	$\lvert S_{ij}\rvert \in]-\infty,+\infty[,\ 1\le i,j\le n$ $\lvert S_{ii}\rvert \le 1,\ 1\le i\le n$	$\surd$	
Zref	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$		50

Description

This function converts a real or complex scattering parameter matrix S into an admittance matrix Y. S has a reference impedance *Zref*, which is assumed to be $Zref = 50\Omega$ if not provided by the user.

Zref can be real or complex number or vector; in the latter case the function operates on the elements of *Zref*.

Example

`Y=stoy(eye(2)*0.1,100)` returns $\begin{bmatrix} 0.00818 & 0 \\ 0 & 0.00818 \end{bmatrix}$.

See also

twoport(), stos(), stoz(), ytos()

stoz()

Converts S-parameter matrix to Z-parameter matrix.

Syntax

Z=stoz(S)

Z=stoz(S, Zref)

Arguments

Name	Type	Def. Range	Required	Default
S	$\mathbb{R}^{n\times n}$, $\mathbb{C}^{n\times n}$	$\lvert S_{ij}\rvert \in \left]-\infty,+\infty\right[,\ 1\le i,j\le n$ $\lvert S_{ii}\rvert \le 1,\ 1\le i\le n$	$\surd$	
Zref	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$\left]-\infty,+\infty\right[$		50

Description

This function converts a real or complex scattering parameter matrix S into an impedance matrix Z. S has a reference impedance *Zref*, which is assumed to be $Zref = 50\Omega$ if not provided by the user.

Zref can be real or complex number or vector; in the latter case the function operates on the elements of *Zref*.

Example

`Z=stoz(eye(2)*0.1,100)` returns $\begin{bmatrix}122 & 0\\ 0 & 122\end{bmatrix}$.

See also

twoport(), stos(), stoy(), ztos()

twoport()

Converts a two-port matrix from one representation into another.

Syntax

U=twoport(X, from, to)

Arguments

Name	Type	Def. Range	Required
X	$\mathbb{R}^{2\times2}$, $\mathbb{C}^{2\times2}$	$]-\infty,+\infty[$	$\surd$
from	Character	$\{'Y', 'Z', 'H', 'G', 'A', 'S', 'T'\}$	$\surd$
to	Character	$\{'Y', 'Z', 'H', 'G', 'A', 'S', 'T'\}$	$\surd$

Description

This function converts a real or complex two-port matrix X from one representation into another.

Example

Transfer a two-port Y matrix Y1 into a Z matrix:

```
Y1=eye(2)*0.1
```

`Z1=twoport(Y1,'Y','Z')` returns $\begin{bmatrix} 10 & 0 \\ 0 & 10 \end{bmatrix}$.

See also

stos(), ytos(), ztos(), stoz(), stoy(), ytoz(), ztoy()

ytos()

Converts Y-parameter matrix to S-parameter matrix.

Syntax

S=ytos(Y)

S=ytos(Y, Z0)

Arguments

Name	Type	Def. Range	Required	Default
Y	$\mathbb{R}^{n\times n}$, $\mathbb{C}^{n\times n}$	$]-\infty,+\infty[$	$\surd$	
Z0	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$		50

Description

This function converts a real or complex admittance matrix Y into a scattering parameter matrix S. Y has a reference impedance *Z0*, which is assumed to be $Z0 = 50\Omega$ if not provided by the user.

Z0 can be real or complex number or vector; in the latter case the function operates on the elements of *Z0*.

Example

`S=ytos(eye(2)*0.1,100)` returns

-0.818	0
0	-0.818

.

See also

twoport(), stos(), ztos(), stoy()

ytoz()

Converts Y-parameter matrix to Z-parameter matrix.

Syntax

Z=ytoz(Y)

Arguments

Name	Type	Def. Range	Required
Y	$\mathbb{R}^{n\times n}$, $\mathbb{C}^{n\times n}$	$]-\infty,+\infty[$	$\surd$

Description

This function converts a real or complex admittance matrix Y into an impedance matrix Z.

Example

`Z=ytoz(eye(2)*0.1)` returns $\begin{bmatrix} 10 & 0 \\ 0 & 10 \end{bmatrix}$.

See also

twoport(), ztoy()

ztos()

Converts Z-parameter matrix to S-parameter matrix.

Syntax

S=ztos(Z)

S=ztos(Z, Z0)

Arguments

Name	Type	Def. Range	Required	Default
Z	$\mathbb{R}^{n\times n}$, $\mathbb{C}^{n\times n}$	$]-\infty,+\infty[$	$\surd$	
Z0	$\mathbb{R}$, $\mathbb{C}$, $\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$		50

Description

This function converts a real or complex impedance matrix Z into a scattering parameter matrix S. Z has a reference impedance *Z0*, which is assumed to be $Z0 = 50\Omega$ if not provided by the user.

Z0 can be real or complex number or vector; in the latter case the function operates on the elements of *Z0*.

Example

`S=ztos(eye(2)*0.1,100)` returns

-0.998	0
0	-0.998

.

See also

twoport(), twoport(), stos(), ytos(), stoz()

ztoy()

Converts Z-parameter matrix to Y-parameter matrix.

Syntax

Y=ztoy(Z)

Arguments

Name	Type	Def. Range	Required
Z	$\mathbb{R}^{n\times n}$, $\mathbb{C}^{n\times n}$	$]-\infty,+\infty[$	$\surd$

Description

This function converts a real or complex impedance matrix Z into an admittance matrix Y.

Example

`Y=ztoy(eye(2)*0.1)` returns $\begin{bmatrix} 10 & 0 \\ 0 & 10 \end{bmatrix}$.

See also

twoport(), ytoz()

15.5.4 Amplifiers

GaCircle()

Circle(s) with constant available power gain Ga in the source plane.

Syntax

y=GaCircle(X,Ga,v)

y=GaCircle(X,Ga,n)

y=GaCircle(X,Ga)

Arguments

Name	Type	Def. Range	Required	Default
X	$\mathbb{R}^{2\times2\times p}$, $\mathbb{C}^{2\times2\times p}$	$]-\infty,+\infty[$	√	
v	$\mathbb{R}^n$	$[0,360]^o$		
Ga	$\mathbb{R}$, $\mathbb{R}^m$	$[0,+\infty[$	√	
n	$\mathbb{N}$	$[2,+\infty[$		64

Description

This function generates the points of the circle of constant available power gain G_A in the complex source plane (r_S) of an amplifier. The amplifier is described by a two-port S-parameter matrix S. Radius r and center c of this circle are calculated as follows:

$$r = \frac{\sqrt{1 - 2\cdot K\cdot g_A\cdot |S_{12}S_{21}| + g_A^2\cdot |S_{12}S_{21}|^2}}{\left|1 + g_A\cdot \left(|S_{11}|^2 - |\Delta|^2\right)\right|} \text{ and } c = \frac{g_A\left(S_{11}^* - S_{22}\,\Delta^*\right)}{1 + g_A\left(|S_{11}|^2 - |\Delta|^2\right)},$$

where $g_A = \dfrac{G_A}{|S_{21}|^2}$ and K Rollet stability factor. Δ denotes determinant of S.

The points of the circle can be specified by the angle vector v, where the angle must be given in degrees. Another possibility is to specify the number n of angular equally distributed points around the circle. If no additional argument to X is given, 64 points are taken. The available power gain can also be specified in a vector Ga, leading to the generation of m circles, where m is the size of Ga.

Please also refer to "Qucs - Technical Papers", chapter 1.5.

Example

```
v=GaCircle(S)
```

See also

GpCircle(), Rollet()

GpCircle()

Circle(s) with constant operating power gain Gp in the load plane.

Syntax

y=GpCircle(X,Gp,v)

y=GpCircle(X,Gp,n)

y=GpCircle(X,Gp)

Arguments

Name	Type	Def. Range	Required	Default
X	$\mathbb{R}^{2\times2\times p}$, $\mathbb{C}^{2\times2\times p}$	$]-\infty,+\infty[$	√	
v	$\mathbb{R}^n$	$[0,360]^o$		
Gp	$\mathbb{R}$, $\mathbb{R}^m$	$[0,+\infty[$	√	
n	$\mathbb{N}$	$[2,+\infty[$		64

Description

This function generates the points of the circle of constant operating power gain G_P in the complex load plane (r_L) of an amplifier. The amplifier is described by a two-port S-parameter matrix S. Radius r and center c of this circle are calculated as follows:

$$r = \frac{\sqrt{1 - 2\cdot K\cdot g_P\cdot |S_{12}S_{21}| + g_P^2\cdot |S_{12}S_{21}|^2}}{\left|1 + g_P\cdot \left(|S_{22}|^2 - |\Delta|^2\right)\right|} \text{ and } c = \frac{g_A\left(S_{22}^* - S_{11}\,\Delta^*\right)}{1 + g_P\left(|S_{22}|^2 - |\Delta|^2\right)},$$

where $g_A = \dfrac{G_P}{|S_{21}|^2}$ and K Rollet stability factor. Δ denotes determinant of S.

The points of the circle can be specified by the angle vector v, where the angle must be given in degrees. Another possibility is to specify the number n of angular equally distributed points around the circle. If no additional argument to X is given, 64 points are taken. The available power gain can also be specified in a vector Gp, leading to the generation of m circles, where m is the size of Gp.

Please also refer to “Qucs - Technical Papers”, chapter 1.5.

Example

```
v=GpCircle(S)
```

See also

GaCircle(), Rollet()

Mu()

Mu stability factor of a two-port S-parameter matrix.

Syntax

y=Mu(S)

Arguments

Name	Type	Def. Range	Required
S	$\mathbb{R}^{2\times2\times p}$, $\mathbb{C}^{2\times2\times p}$,$\mathbb{R}^{2\times2}$, $\mathbb{C}^{2\times2}$	$]-\infty,+\infty[$	$\surd$

Description

This function returns the Mu stability factor μ of an amplifier being described by a two-port S-parameter matrix S:

$$\mu = \frac{1-|S_{11}|^2}{|S_{22}-S_{11}^{*}\,\Delta| + |S_{21}\,S_{12}|}$$

Δ denotes determinant of S.

The amplifier is unconditionally stable if $\mu > 1$.

For S being a vector of matrices the equation above is applied to the sub-matrices of S.

Example

```
m=Mu(S)
```

See also

Mu2(), Rollet(), StabCircleS(), StabCircleL()

Mu2()

Mu' stability factor of a two-port S-parameter matrix.

Syntax

y=Mu2(S)

Arguments

Name	Type	Def. Range	Required
S	$\mathbb{R}^{2\times2\times p}$, $\mathbb{C}^{2\times2\times p}$,$\mathbb{R}^{2\times2}$, $\mathbb{C}^{2\times2}$	$]-\infty,+\infty[$	$\surd$

Description

This function returns the Mu' stability factor μ' of an amplifier being described by a two-port S-parameter matrix S:

$$\mu' = \frac{1-|S_{22}|^2}{|S_{11}-S_{22}^*\,\Delta|+|S_{21}\,S_{12}|}$$

Δ denotes determinant of S.

The amplifier is unconditionally stable if $\mu' > 1$.

For S being a vector of matrices the equation above is applied to the sub-matrices of S.

Example

```
m=Mu2(S)
```

See also

Mu2(), Rollet(), StabCircleS(), StabCircleL()

NoiseCircle()

Generates circle(s) with constant Noise Figure(s).

Syntax

y=NoiseCircle(Sopt,Fmin,Rn,F,v)

y=NoiseCircle(Sopt,Fmin,Rn,F,n)

y=NoiseCircle(Sopt,Fmin,Rn,F)

Arguments

Name	Type	Def. Range	Required	Default
Sopt	$\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	√	
Fmin	$\mathbb{R}^n$	$[1,+\infty[$	√	
Rn	$\mathbb{R}^n$, $\mathbb{C}^n$	$[0,+\infty[$	√	
F	$\mathbb{R}$, $\mathbb{R}^n$	$[1,+\infty[$	√	
v	$\mathbb{R}^n$	$[0,360]^o$		
n	$\mathbb{N}$	$[2,+\infty[$		64

Description

This function generates the points of the circle of constant Noise Figure (NF) F in the complex source plane (r_S) of an amplifier. Generally, the amplifier has its minimum NF F_{min}, if the source reflection coefficient $r_S = S_{opt}$(noise matching). Note that this state with optimum source reflection coefficient S_{opt} is different from power matching ! Thus power gain under noise matching is lower than the maximum obtainable gain. The values of S_{opt}, $F_{min}$and the normalised equivalent noise resistance R_n/Z_0can be usually taken from the data sheet of the amplifier.

Radius r and center c of the circle of constant NF are calculated as follows:

$$r = \frac{\sqrt{N^2 + N \cdot \left(1 - |S_{opt}|^2\right)}}{1+N} \text{ and } c = \frac{S_{opt}}{1+N}, \text{ with } N = \frac{F - F_{min}}{4\,R_n} \cdot Z_0 \cdot |1 + S_{opt}|^2 \ .$$

The points of the circle can be specified by the angle vector v, where the angle must be given in degrees. Another possibility is to specify the number n of angular equally distributed points around the circle. If no additional argument to X is given, 64 points are taken.

Please also refer to “Qucs - Technical Papers”, chapter 2.2.

Example

```
v=NoiseCircle(Sopt,Fmin,Rn,F)
```

See also

GaCircle(), GpCircle()

PlotVs()

Returns a data item based upon vector or matrix vector with dependency on a given vector.

Syntax

y=PlotVs(X, v)

Arguments

Name	Type	Def. Range	Required
X	$\mathbb{R}^n$, $\mathbb{C}^n$, $\mathbb{R}^{m\times n\times p}$, $\mathbb{C}^{m\times n\times p}$	$]-\infty,+\infty[$	$\surd$
v	$\mathbb{R}^n$, $\mathbb{C}^n$	$]-\infty,+\infty[$	$\surd$

Description

This function returns a data item based upon a vector or matrix vector X with dependency on a given vector v.

Example

`PlotVs(Gain,frequency/1E9)`.

See also

Rollet()

Rollet stability factor of a two-port S-parameter matrix.

Syntax

y=Rollet(S)

Arguments

Name	Type	Def. Range	Required
S	$\mathbb{R}^{2\times2\times p}$, $\mathbb{C}^{2\times2\times p}$,$\mathbb{R}^{2\times2}$, $\mathbb{C}^{2\times2}$	$]-\infty,+\infty[$	$\surd$

Description

This function returns the Rollet stability factor K of an amplifier being described by a two-port S-parameter matrix S:

$$K = \frac{1 - |S_{11}|^2 - |S_{22}|^2 + |\Delta|^2}{2\,|S_{21}|\,|S_{12}|}$$

Δ denotes determinant of S.

The amplifier is unconditionally stable if $K > 1$ and $|\Delta| < 1$.

Note that a large K may be misleading in case of a multi-stage amplifier, pretending extraordinary stability. This is in conflict with reality where a large gain amplifier usually suffers from instability due to parasitics.

For S being a vector of matrices the equation above is applied to the sub-matrices of S.

Example

```
K=Rollet(S)
```

See also

Mu(), Mu2(), StabCircleS(), StabCircleL()

StabCircleL()

Stability circle in the load plane.

Syntax

y=StabCircleL(X)

y=StabCircleL(X,v)

y=StabCircleL(X,n)

Arguments

Name	Type	Def. Range	Required	Default
X	$\mathbb{R}^{2\times2\times p}$, $\mathbb{C}^{2\times2\times p}$	$]-\infty,+\infty[$	√	
v	$\mathbb{R}^n$	$[0,360]^o$		
n	$\mathbb{N}$	$[2,+\infty[$		64

Description

This function generates the stability circle points in the complex load reflection coefficient (r_L) plane of an amplifier. The amplifier is described by a two-port S-parameter matrix S. Radius r and center c of this circle are calculated as follows:

$$r = \left|\frac{S_{21}\,S_{12}}{|S_{22}|^2 - |\Delta|^2}\right| \text{ and } c = \frac{S_{22}^* - S_{11}\cdot\Delta^*}{|S_{22}|^2 - |\Delta|^2}$$

Δ denotes determinant of S.

The points of the circle can be specified by the angle vector v, where the angle must be given in degrees. Another possibility is to specify the number n of angular equally distributed points around the circle. If no additional argument to X is given, 64 points are taken.

If the center of the $r_L$plane lies within this circle and $|S_{11}| \leq 1$ then the circuit is stable for all reflection coefficients inside the circle. If the center of the $r_L$plane lies outside the circle and $|S_{11}| \leq 1$ then the circuit is stable for all reflection coefficients outside the circle (please also refer to "Qucs - Technical Papers", chapter 1.5).

Example

```
v=StabCircleL(S)
```

See also

StabCircleS(), Rollet(), Mu(), Mu2()

StabCircleS()

Stability circle in the source plane.

Syntax

y=StabCircleS(X)

y=StabCircleS(X,v)

y=StabCircleS(X,n)

Arguments

Name	Type	Def. Range	Required	Default
X	$\mathbb{R}^{2\times2\times p}$, $\mathbb{C}^{2\times2\times p}$	$]-\infty,+\infty[$	$\surd$	
v	$\mathbb{R}^n$	$[0,360]^o$		
n	$\mathbb{N}$	$[2,+\infty[$		64

Description

This function generates the stability circle points in the complex source reflection coefficient (r_S) plane of an amplifier. The amplifier is described by a two-port S-parameter matrix S. Radius r and center c of this circle are calculated as follows:

$$r = \left|\frac{S_{21}\,S_{12}}{|S_{11}|^2 - |\Delta|^2}\right| \text{ and } c = \frac{S_{11}^* - S_{22}\cdot\Delta^*}{|S_{11}|^2 - |\Delta|^2}$$

Δ denotes determinant of S.

The points of the circle can be specified by the angle vector v, where the angle must be given in degrees. Another possibility is to specify the number n of angular equally distributed points around the circle. If no additional argument to X is given, 64 points are taken.

If the center of the $r_S$plane lies within this circle and $|S_{22}| \leq 1$ then the circuit is stable for all reflection coefficients inside the circle. If the center of the $r_S$plane lies outside the circle and $|S_{22}| \leq 1$ then the circuit is stable for all reflection coefficients outside the circle (please also refer to "Qucs - Technical Papers", chapter 1.5).

Example

```
v=StabCircleS(S)
```

See also

StabCircleL(), Rollet(), Mu(), Mu2()

StabFactor()

Stability factor of a two-port S-parameter matrix. Synonym for Rollet()

Syntax

y=StabFactor(S)

See also

Rollet()

StabMeasure()

Stability measure B1 of a two-port S-parameter matrix.

Syntax

y=StabMeasure(S)

Arguments

Name	Type	Def. Range	Required
S	$\mathbb{R}^{2\times2\times p}$, $\mathbb{C}^{2\times2\times p}$,$\mathbb{R}^{2\times2}$, $\mathbb{C}^{2\times2}$	$]-\infty,+\infty[$	$\surd$

Description

This function returns the stability measure *B1* of a two-port S-parameter matrix S:

$$B1 = 1 + |S_{11}|^2 - |S_{22}|^2 - |\Delta|^2$$

Δ denotes determinant of S.

The amplifier is unconditionally stable if $B1 > 0$ and the Rollet factor $K > 1$.

For S being a vector of matrices the equation above is applied to the sub-matrices of S.

Example

```
B1=StabMeasure(S)
```

See also

Rollet(), Mu(), Mu2(), StabCircleS(), StabCircleL()

vt()

Thermal voltage for a given temperature in Kelvin.

Syntax

y=vt(t)

Arguments

Name	Type	Def. Range	Required	Default
t	$\mathbb{R}$	$[0, +\infty[$	$\surd$	

Description

This function returns the corresponding thermal voltage V_t in Volt of a real absolute temperature (vector) T in Kelvin according to

$$V_t = \frac{kT}{e}$$

where k is the Boltzmann constant and e denotes the electrical charge on the electron. For t being a vector the equation above is applied to the components of k.

Please note that t is always larger than or equal to zero.

Example

`y=vt(300)` returns 0.0259.

16 Component, compact device and circuit modelling using symbolic equations

16.1 Introduction

Qucs releases 0.0.11 and 0.0.12 mark a turning point in the development of the Qucs component and circuit modelling facilities. Release 0.0.11 introduced component values defined by equations and for the first time allowed subcircuits with parameters. Release 0.0.12 extends these features to add model development using symbolic equations that are similar to compact device code written in the Verilog-A modelling language. In designing the latest Qucs modelling features the Qucs team has made a central focus of their work the need to provide the package with an interactive and easy to use modelling system which allows fast model prototype construction. Much of these new aspects have up to now been undocumented and are likely to be very new to most Qucs users. The aim of this tutorial note is to outline the background to these important package extensions and to provide real help to Qucs users who are interested in writing and experimenting with their own models. The text includes a number of illustrative examples for readers to try and experiment with.

16.2 Qucs electronic device and circuit modelling

Circuit simulation packages are complex software systems which often take years to mature to a stage where they are capable of analysing the current generation of integrated and discrete electronic circuits. Most circuit simulators have a number of common basic attributes; firstly circuits are represented by a textual netlist or a schematic diagram which contains all the information required by a simulator to analyse the performance of a circuit, and secondly a simulation engine which undertakes the calculation of circuit performance in one or more different circuit domains such as DC, AC or transient, and thirdly a post simulation processing system which structures and displays the simulation data in both

tabular and graphical forms. All circuit simulators have one other important attribute, namely that they represent individual electronic components by a model, or abstraction, in a way that can be understood and analysed by the simulation engine when undertaking a simulation task. Without component models the science of circuit simulation would not have developed to the stage it has today. From a users point of view component models are the key to simulator productivity; the greater the number of different models the easier it becomes to analyse mixed analogue and digital electronic systems.

Shown in Fig. 16.1 is a block diagram of the analogue component modelling and simulation facilities currently provided by the Qucs package. The diagram is structured as a flow chart which emphasises the different device modelling routes. When Qucs was first released only two of these were available for users to develop new device models. The first of these has been used extensively by the package developers to construct the built-in models that are distributed with each Qucs release. This fundamental route involves hand coding the C++ code for a new model[1], its compilation and linking with the core Qucs C++ code. Obviously, this does require a specialised knowledge of the Qucs model programming interface[2], the necessary C++ skills, including a good working knowledge of the Trolltech Qt toolkit[3]. At the time of writing these notes the latest device to be added to Qucs using this approach is the exponential pulse source[4]. Models based on hand written C++ code are normally restricted to basic devices that form the fundamental component core of a simulator - particularly where simulation computational efficiency is important. One disadvantage of this approach, is the obvious one, in that the time to implement a new model increases disproportionately with increasing model complexity. For most Qucs users this route would not be the most natural to use when developing new models. However, for the specialist who spends a significant amount of time researching new device models this has always in the past, been the route of choice. Unfortunately, modern semiconductor device models are becoming so complex that the model development time can stretch into months or even years and requires typically thousands of lines of C or C++ code to characterise a model[5]. With the more complex models the problem of finding bugs in the model code also acts as a limit to fast model development.

For the average Qucs user their first introduction to the software is probably through constructing circuit schematics made entirely from the standard component models built

[1]The technical details of the built-in models are described in: Qucs Technical Papers, Stefan Jahn, Michael Margraf, Vincent Habchi and Raimund Jacob, `http://qucs.sourceforge.net/technical.html`.

[2]Writing the documentation for the Qucs model programming interface is on the to do list and will be completed, when time allows, sometime in the future.

[3]Qt is a registered trademark of Trolltech, Norway; `http://www.trolltech.com/copyright`.

[4]Added by Gunther Kraut on 15 April 2007. This device has been added for compatibility with SPICE.

[5]A good introduction to writing compact device models is given in "How to (and how not to) write a compact model in Verilog-A", Geoffrey J. Coram, 2004, Proc. 2004 IEEE International Behavioral Modeling and Simulation Conference (BMAS 2004), pp 97- 106.

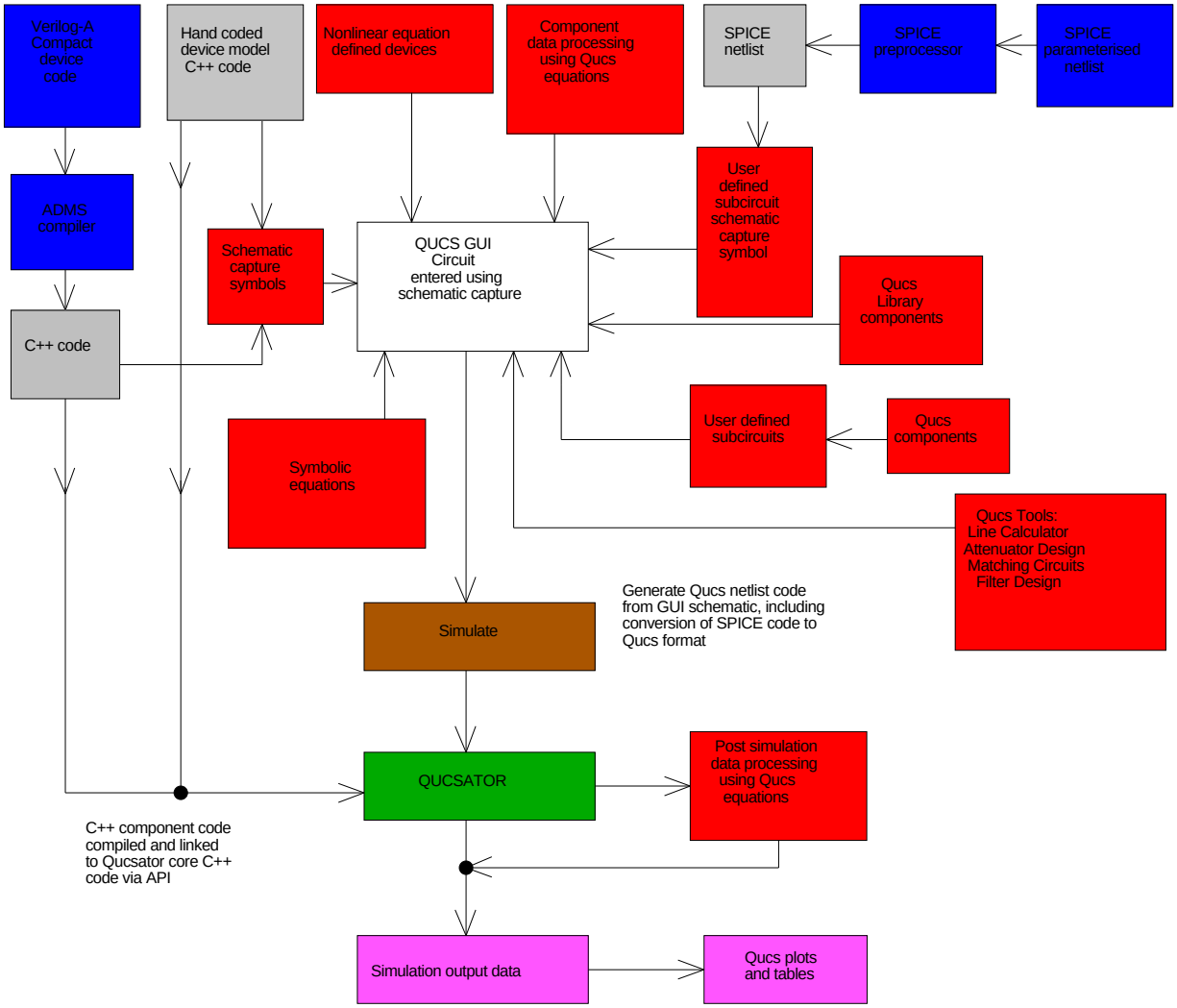


Figure 16.1: Qucs analogue component modelling and simulation block diagram (not including optimisation)

into the package and the testing of their performance by launching the simulator from one of the Qucs simulation icons.[6] The next natural stage in the Qucs modelling and simulation learning curve is the use of subcircuits where groups of built-in components are collected together to form a higher level circuit block. These blocks are often arranged with a common theme, forming a Qucs library. The process of modelling new devices/circuits is normally done by connecting existing component models and user defined subcircuits. With this type of modelling higher level functional models can only be constructed from existing fundamental components or previously constructed subcircuits. Engineers often call this approach to modelling, macromodelling. Qucs releases up to 0.0.10 relied on macromodelling for functional model development via the Qucs schematic interface. This route remains popular amongst most Qucs users because it is easy to understand, is fully interactive and allows straight forward testing of new models. One feature that is common to all components included in Qucs releases up to 0.0.10 may not be immediately obvious to readers, namely that, with the exception of sweep variables, component values could only be numbers, for example R1 = 1k, and were not allowed to be represented by algebraic expressions like R1 = Value1, where Value1 = $100.0+50 \cdot \mathrm{X}$. Its also worth pointing out at this point that during simulation, again performed by Qucs releases up to 0.0.10, component values were required to remain constant and could not be a function of the circuit variables such as voltage, current or charge.

One way to remove the component value restrictions imposed by early Qucs releases is to model devices and circuits using preprocessor extended forms of the SPICE netlist language. Circuit design equations can then be embedded in SPICE netlists and the calculation of component values completed by the SPICE preprocessor. Both the SPICE to Qucs and OP AMP tutorials[7] outline in detail the steps required to merge circuit design and simulation in this way. This modelling route is a very important and powerful model development tool. So much so that ongoing tests to identify how compatible Qucs is with the industrial standard SPICE 2g6 and 3f5 syntax are currently being undertaken as part of the Qucs development schedule[8]. Although perfectly viable as a model development tool the use of an extended SPICE netlist language has a number of serious disadvantages, namely that not all the Qucs built-in component models have equivalent SPICE models and secondly text netlists are the only entry medium for describing models.

The previous paragraphs give a brief statement of the different component modelling routes that were available up to release 0.0.11. Qucs 0.0.11 is very much a modelling water shed in

[6]The "Getting Started with Qucs" tutorial by Stefan Jahn outlines a number of basic simulation techniques; `http://qucs.sourceforge.net/docs.html`.

[7]Qucs simulation of SPICE netlists and Modelling Operational Amplifiers, Mike Brinson, `http://qucs.sourceforge.net/docs.html`.

[8]Qucs: Report Book; SPICE to Qucs test reports, Mike Brinson, `http://qucs.sourceforge.net/docs.html`.

that symbolic equations were introduced for the calculation of component values, previously equations were only allowed when structuring simulation output data for post simulation listing or plotting. Release 0.0.11 allows the following types of variable;

1. sweep variables,
2. equations left hand side,
3. component parameter's left hand side (e.g. R1.R),
4. subcircuit parameters and
5. simulation output data.

With each Qucs release the number of analysis functions, and other data processing features, included in the Qucs equation set continues to expand[9]. From release 0.0.11 parameters are also allowed with subcircuits so that data can be passed to a model. This allows generalised subcircuit/macromodels to be developed for popular devices such as operational amplifiers. Through the use of embedded design equations within subcircuits and parameter passing it became possible to construct powerful models that mix both circuit design procedures and the calculation of individual component values. Qucs 0.0.11 still imposed the restriction that equations could not be functions of voltage, current or charge.

With the release of Qucs 0.0.12 the voltage, current and charge restrictions imposed on equations will finally be relaxed. The introduction of a new device modelling component called the equation defined device (EDD) allows firstly device current to be formulated as a function of voltage, and secondly device charge to be calculated as a function of voltage and current. The syntax adopted for the new model borrows heavily on the compact device modelling approach taken by the Verilog-A modelling language.

Some readers will probably have noted that so far these notes make no reference to the ADMS model development route illustrated in Fig. 16.1. ADMS stands for Automated device model synthesizer[10] and includes a Verilog-A to C/C++ compiler. It allows compact device models to be described in the Verilog-A language then compiled to C/C++ and the resulting code linked with the Qucs core simulation code[11]. Model development using ADMS is similar to the fundamental hand coded C++ model development route except that model development is greatly simplified by the power of the high level Verilog-A language. A strong relationship exists between the ADMS and EDD modelling procedures in that

[9] See Measurement Expressions Reference Manual, Gunther Kraut and Stefan Jahn, `http://qucs.sourceforge.net/docs.html`.

[10] L.Lemaitre, C.C. McAndrew, and S. Hamm, ADMS - Automated Device Model Synthesizer, Proc. IEEE CICC, 2002.

[11] For more details see, Qucs Description: Verilog-AMS interface, Stefan Jahn and Hélène Parruitte, `http://qucs.sourceforge.net/docs.html`.

EDD can be considered a fast interactive model prototyping method whose equations can easily be expressed in Verilog-A and compiled into C/C++ code for permanent inclusion in the Qucs simulator[12].

The opening paragraphs attempt to outline the available device modelling techniques that are central to the functioning of the Qucs package. The remaining sections of this tutorial note are devoted to illustrating the power of Qucs modelling through the introduction of a number of illustrative examples. Initially these start from a simple, and hopefully familiar, point and then proceed to more complex examples which present many of the concepts lightly touched upon in the opening text.

16.3 Extending circuit simulation capabilities with equations

Just adding component value calculations, via equations, to a circuit simulator immediately increases the underlying design and simulation capabilities way beyond that found in earlier generation simulators. Consider the simple RC circuit shown in Fig. 16.2. Capacitor *Cap* is stepped from 0.1µF to 1.1µF and the small signal AC response of the network calculated. In this example the values for both *R1* and *Cap* are given as numeric values. The simulation test shows the effect of stepping the value of one component through a series of values and recording the effect of component changes on circuit performance. In other words this is a classical circuit analysis use of a circuit simulator. In a real design situation different data is often required. Most designers would prefer to find the value of *Cap* that gives a specific RC cut-off frequency (f_c) for a specified value of *R1*. This is the type of investigative problem where adding equations into the simulation process generates more informative results. Shown in Fig. 16.3 is a similar RC network to that illustrated in Fig. 16.2.

Capacitor voltage $VCap$ is given by:

$$V_{Cap} = \frac{V_1}{\sqrt{1 + \omega^2 \cdot C_1^2 \cdot R_1^2}} \tag{16.1}$$

where the cut-off frequency in the voltage transfer function is

$$f_c = \frac{1}{2\pi \cdot R_1 \cdot C_1} \tag{16.2}$$

Hence, by expressing *Cap* as a function of f_c and stepping f_c through a range of frequencies, the effect of capacitance changes on the voltage transfer function can be found. More

[12]Appendix A gives an operator and function comparison table for Qucs and Verilog-A.

importantly a nomogram of *Cap* values against f_c can be plotted giving the circuit designer a visual aid for determing the value of *Cap* required for given values of *R1* and f_c. Although the circuits shown in Figs. 16.2 and 16.3 are very basic they do demonstrate how much more powerful a circuit simulator becomes when component values are calculated using equations.

16.3.1 Low pass active filter design with embedded design equations

In this section a more advanced circuit design example is introduced to illustrate the power of embedded design equations in a Qucs simulation schematic. A second order Sallen-Key low pass filter is employed for this task because it is so well known and most readers are likely to have met it's design in the past. A second order low pass filter is represented by the voltage transfer function:

$$A(S) = \frac{V_{out}}{V_{in}} = \frac{A0}{(1 + a_2 \cdot S + b_2 \cdot S^2)} \tag{16.3}$$

where $A0$ is the passband DC gain and coefficients a_2, b_2 are for Bessel, Butterworth, Tschebyscheff or similar polynomials.

The following list[13] gives the second order coefficients for the Bessel → 1.3617, 0.618; Butterworth → 1.4142, 1.000; and 3dB ripple Tschebyscheff → 1.065, 1.9305, polynomials. The second order Sallen-Key low pass filter circuit is shown in Fig. 16.4. This circuit has a voltage gain transfer function given by:

$$A(S) = \frac{A0}{1 + \omega_c \cdot \left[C_1 \cdot (R_1 + R_2) + (1 - A0) \cdot R_1 \cdot C_2\right] \cdot S + \omega_c^2 \cdot R_1 \cdot R_2 \cdot C_1 \cdot C_2 \cdot S^2} \tag{16.4}$$

where

$$A0 = 1 + \frac{R_3}{R_4} \tag{16.5}$$

This can be simplified by letting $R_1 = R_2 = R$ and $C_1 = C_2 = C$; the transfer function then becomes:

[13] See OP Amps for everyone, Chapter 16: Active filter design technology, Texas Instruments, August 2002, SL0D006B, PP 16.1,16.63.

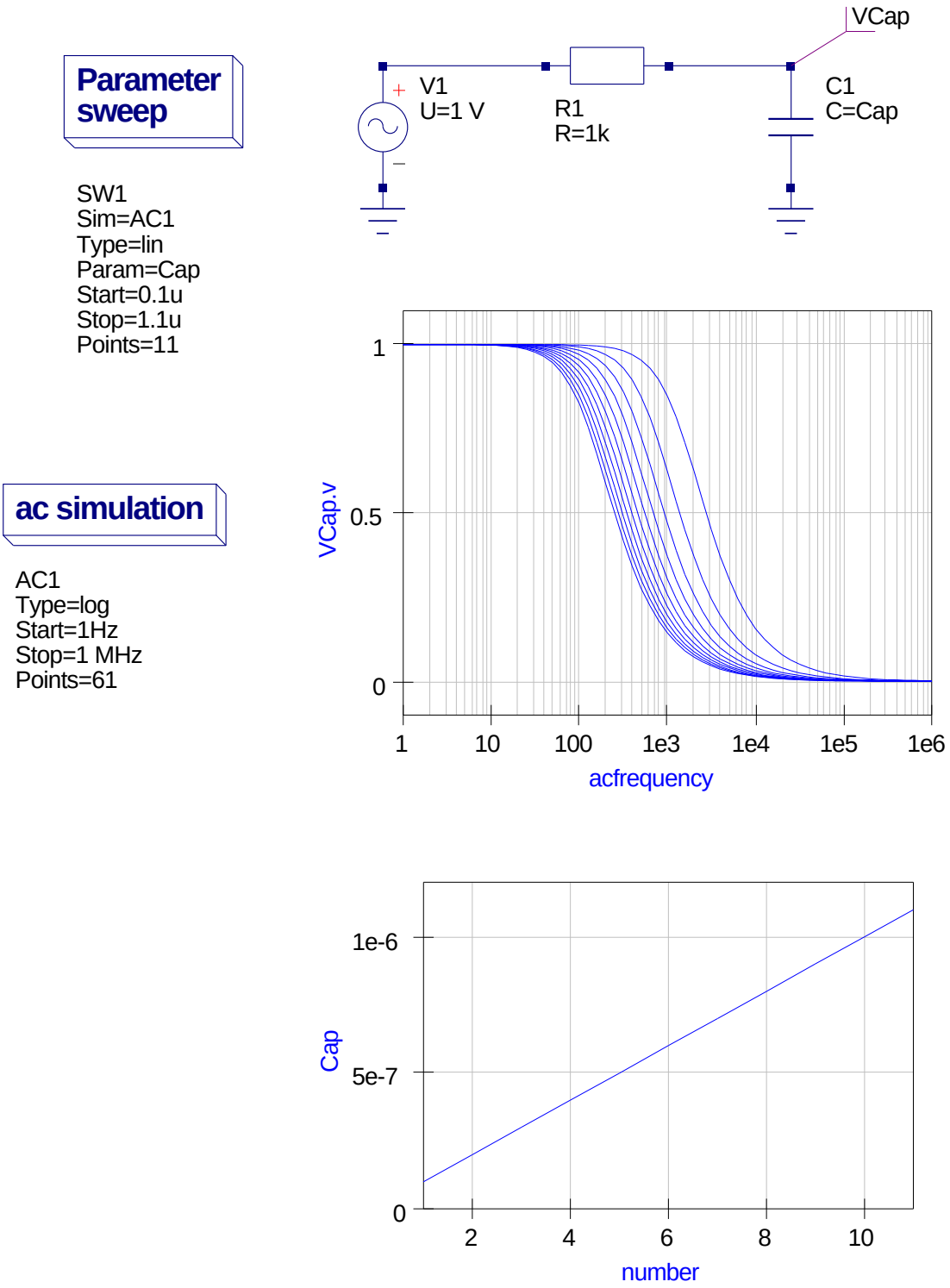


Figure 16.2: A simple RC circuit simulation using numerical component values

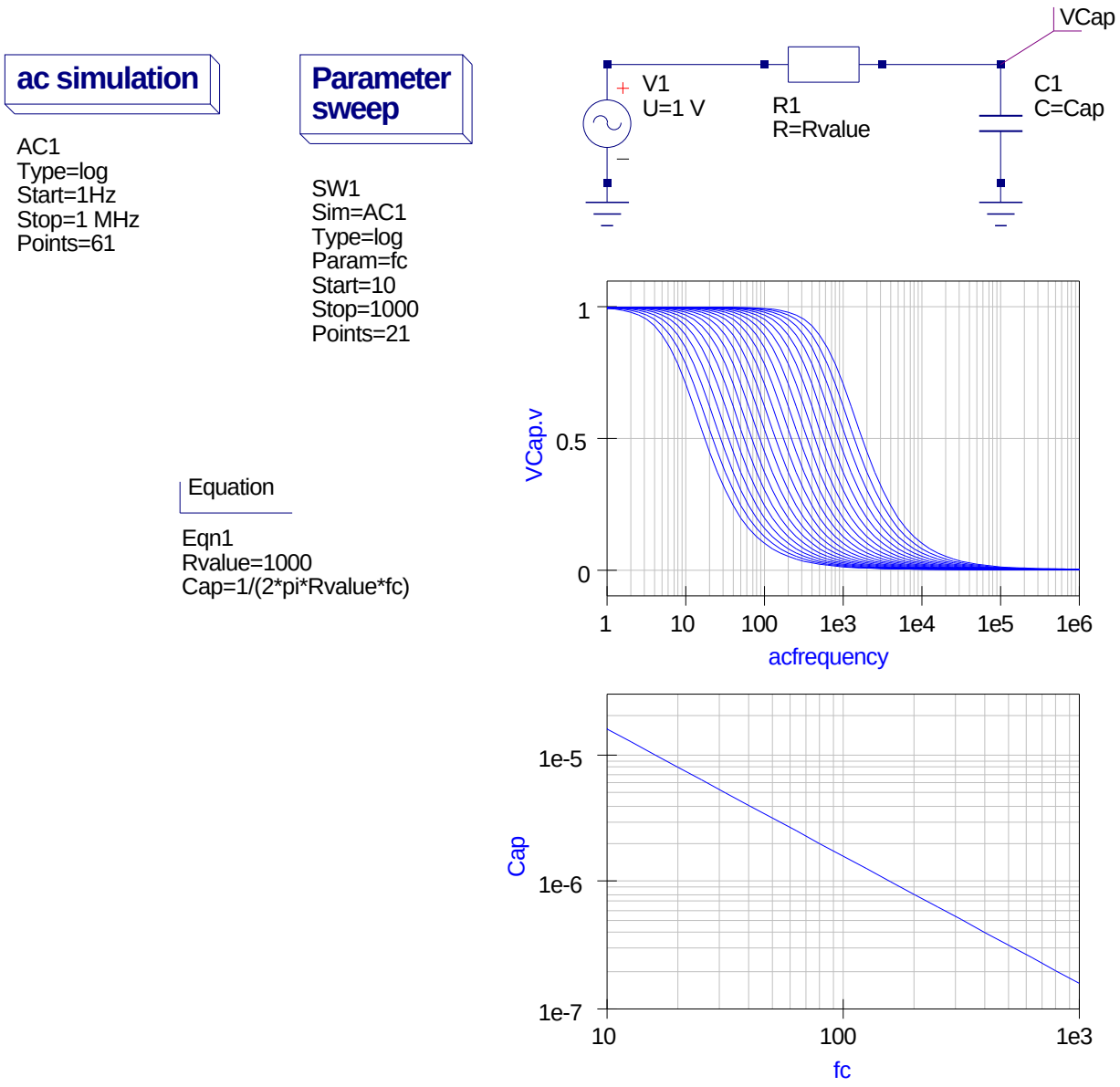


Figure 16.3: A simple RC circuit simulation employing equation determined component values

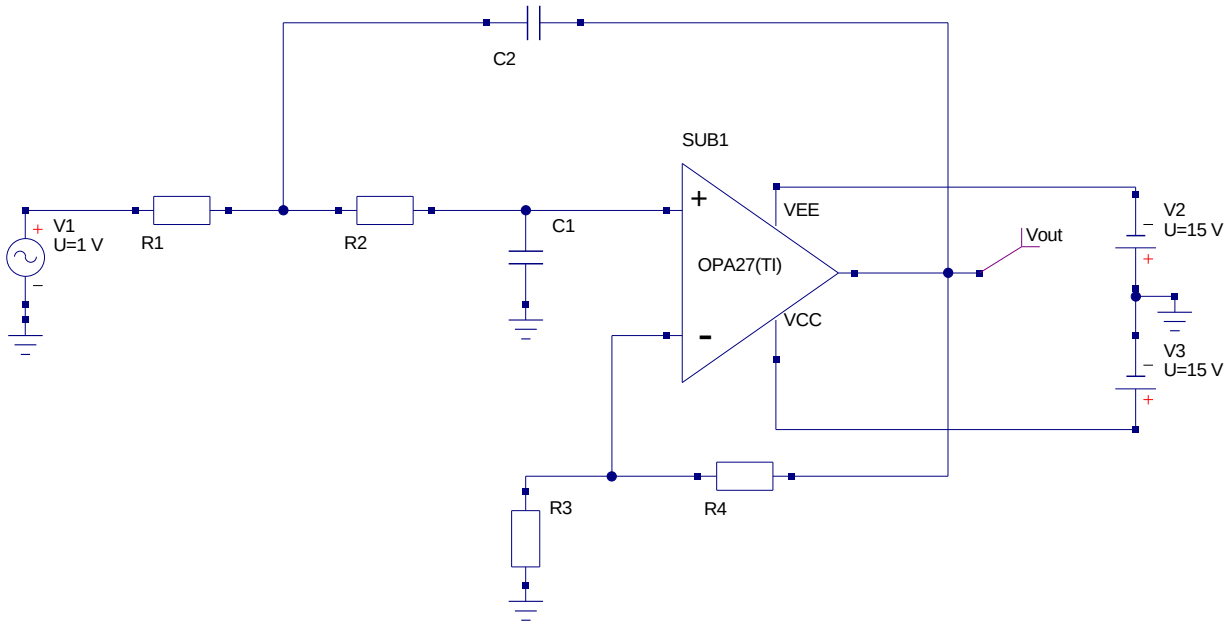


Figure 16.4: The Sallen-Key lowpass active filter circuit

$$A(S) = \frac{A0}{1 + [\omega_c \cdot R \cdot C \cdot (3 - A0)] \cdot S + \left[(\omega_c \cdot R \cdot C)^2\right] \cdot S^2}. \tag{16.6}$$

By comparison

$$a_2 = \omega_c \cdot R \cdot C \cdot (3 - A0) \tag{16.7}$$

and

$$b_2 = (\omega_c \cdot R \cdot C)^2 \tag{16.8}$$

Fixing C and solving for R and $A0$, yields

$$R = \frac{\sqrt{b_2}}{\omega_c \cdot C} \text{ , and } A0 = 3 - \frac{a_2}{\sqrt{b_2}}. \tag{16.9}$$

Also once A0 is known the value for $R4$ can be calculated using equation

$$A0 = 1 + \frac{R3}{R4}. \tag{16.10}$$

Hence by providing values for C and $R3$ the values for R and $A0$, and of course $R4$, can be determined for a specified cut off frequency fc. Figure 16.5 shows the final design schematic and the simulation results for this example. A number of important observations can be made from Fig. 16.5:

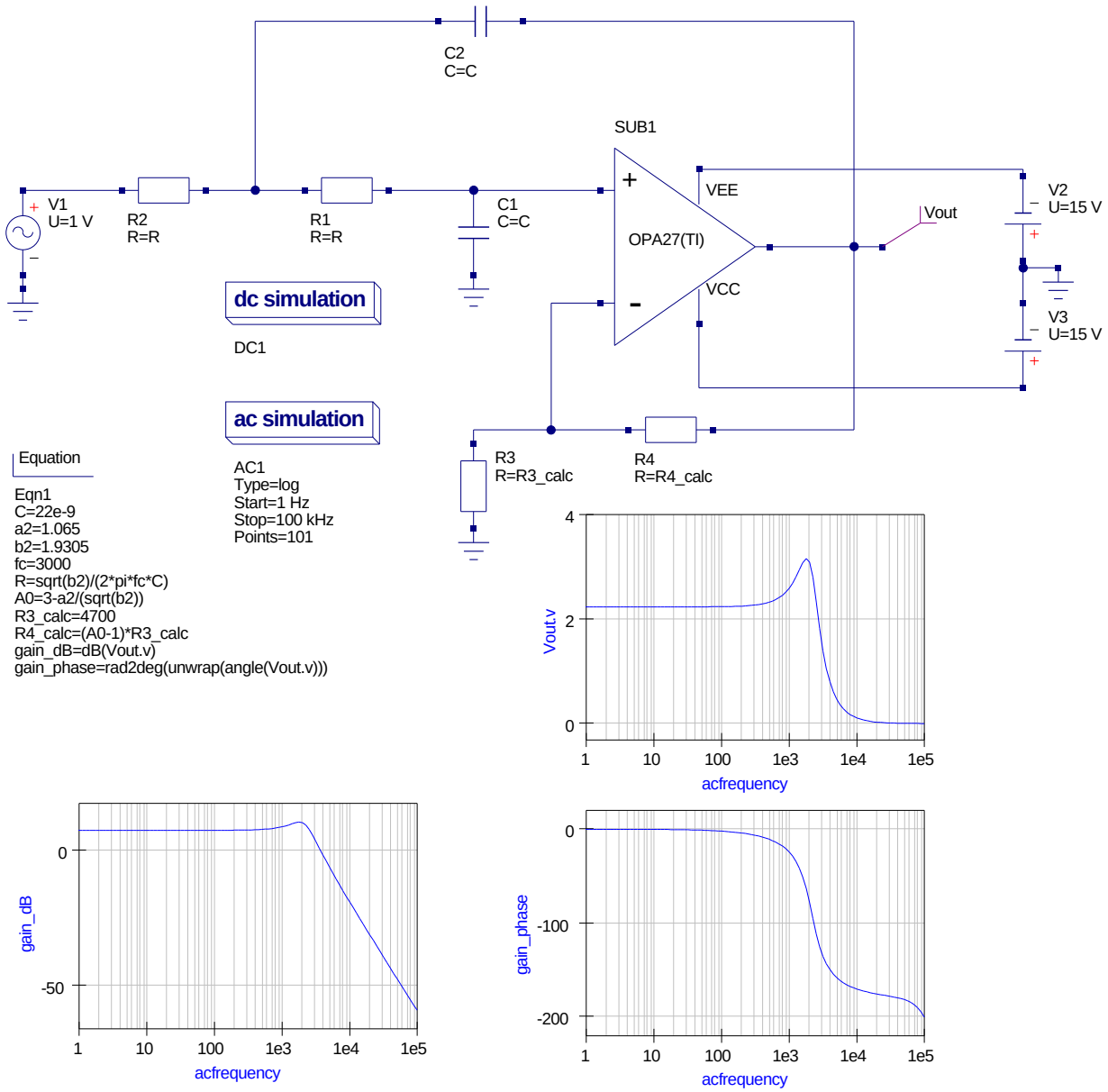


Figure 16.5: The Sallen-Key lowpass active filter schematic with embedded design equations

1. One or more equation blocks hold both design and post simulation data processing equations plus assignments for named items: C, fc and $R3$ are given numerical values, the a and b polynomial coefficients are set to the values introduced in the text, and finally the design equations for R, $A0$ and $R4$ calculations are listed.
2. The order of entries in equation blocks is not important because Qucs automatically sorts out the data it requires when calculating equations.
3. The lefthand quantities in the assignment entries in the equation blocks are linked to the component values in the schematic, see for example C and R.
4. The OP27 operational amplifier model is from the modified Qucs 0.0.11 OPAMP library. This model was generated using the SPICE to Qucs modelling route.
5. To design and simulate a Sallen-Key low pass filter with a different cut off frequency[14] simply change the value of fc and rerun the Qucs simulator.
6. On completion of a simulation, pressing key F5 (Show last messages) causes the simulation log to be displayed. This includes the calculated values of the components and the netlist for the circuit, see Fig. 16.6.
7. One final point of significance that some readers may have noticed - all numerical values in equation blocks must be specified in scientific notation; electronic notation like 1k or 3nF is not allowed[15].

[14] If the design calculations result in impractical values for the filter components then the value of C should be changed and the simulation repeated.

[15] In long term it is expected that electronic notation will be allowed. The changes for this are on the to do list but at the moment the work has a low priority.

```
Output:
--------
netlist content
  13 R instances
   5 C instances
   2 VCCS instances
   5 CCCS instances
   2 VCVS instances
   1 CCVS instances
   8 Vdc instances
   1 Idc instances
   1 Vac instances
   4 Diode instances
   2 BJT instances
   1 DC instances
   1 AC instances
creating netlist...
checker notice, variable 'Vout.v' in equation 'gain_dB' not yet defined
checker notice, variable 'Vout.v' in equation 'gain_phase' not yet defined
kB = 1.38065e-23
e = 2.71828
pi = 3.14159
C = 2.2e-08
a2 = 1.065
b2 = 1.9305
fc = 3000
R = 3350.51
A0 = 2.2335
R3_calc = 4700
R4_calc = 5797.43
kB = 1.38065e-23
e = 2.71828
pi = 3.14159
kB = 1.38065e-23
e = 2.71828
pi = 3.14159
kB = 1.38065e-23
e = 2.71828
pi = 3.14159
```

Figure 16.6: Message output log for the simulation of the Sallen-key low pass circuit: for brevity only the component value section is given

16.4 Introduction to Qucs subcircuit parameters

Subcircuits are a concept that has been part of the simulation scene for a long time. All circuit simulators based on SPICE have subcircuits as part of their basic device compliment. This is not surprising because they form a natural way of breaking an electronic system down into a number of smaller self contained functional blocks. What is surprising however, is the fact that a significant number of simulators, including SPICE 2g6 and 3f5[16], do not allow parameters to be passed to a subcircuit. Parameter passing appears to have been first introduced when a number of the popular commercial circuit simulators were being developed[17]. Qucs releases up to version 0.0.10 are similar to SPICE in that they also did not allow parameters with subcircuits.

This very important limitation has been removed with release 0.0.11, which allows parameters to be attached to component symbols and used in subcircuit equation calculations. Shown in Fig. 16.7 are the circuit schematic and user generated symbol for a simple harmonic generator with a fundamental and three harmonic sinusoidal components. Parameters *f1* to *f4* determine these frequency components. Notice that an equation block, at the circuit schematic level, is used to calculate the harmonic frequencies. Parameters *ph1* to *ph4* set the phase of the individual sinusoidal oscillators. The process of attaching parameters, and their default values, to a subcircuit symbol is straightforward; simply right click on the symbol subcircuit name, *SUB1* in Fig. 16.7, and an Edit Subcircuit Properties dialog box appears allowing parameter names and their default values to be entered[18]. Subcircuit parameters and their values are normally displayed as a list underneath the subcircuit name. Changing parameter values is done in a similar fashion to changing the values of the standard built-in components. The diagram and simulation results illustrated in Fig. 16.8 show a waveform formed from a fundamental and two harmonics.

An equation block is employed to calculate and plot the amplitude and power spectral densities of the harmonic waveform. By changing the fundamental frequency, signal amplitudes and phases different wave shapes can be generated by resimulating the circuit. In this example transient analysis is used to generate the harmonic waveform with the run time set to 10ms and the number of points equal to 500[19]. This gives a sampling time of 20µs and a sampling frequency of 50kHz. Equation block *Eqn1* demonstrates how the Qucs functions[20] can be used to postprocess simulation generated data - in this example they

[16]One of the reasons SPICE preprocessors were developed was to allow parameter passing to subroutines, for more details see Qucs Tutorial: Qucs simulation of SPICE netlists, Mike Brinson, `http://qucs.sourceforge.net/`.

[17]See, for example, the extended netlist format originally designed by the MicroSim Corporation for the PSpice circuit simulator.

[18]See Appendix B for a more detailed description of the procedure.

[19]Qucs function length() determines the correct data length in equation block Eqn1 calculations.

[20]If you have used a program like Octave, or indeed Matlab, many of these functions should be familiar

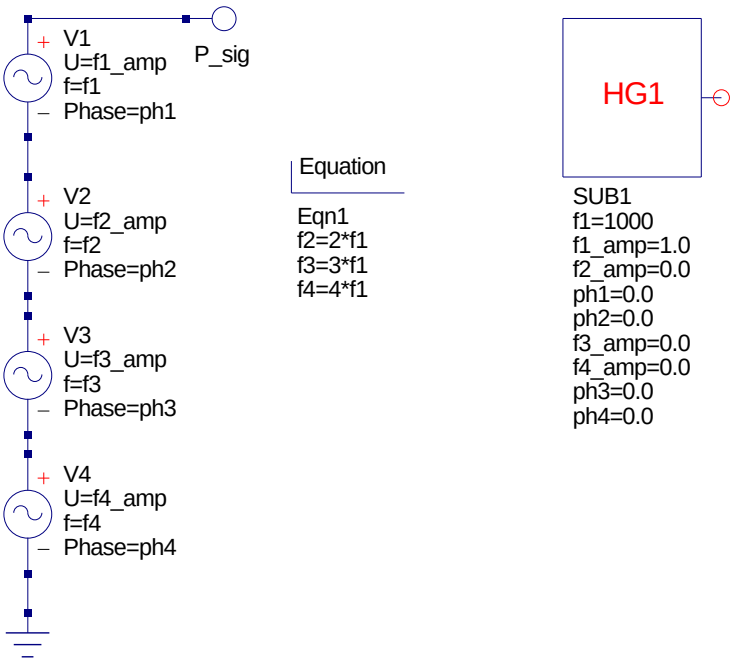


Figure 16.7: Harmonic generator subcircuit schematic and symbol

are used to compute the DFT of the harmonic generator waveform, convert the resulting spectra from double sided to single sided form, compute and plot the amplitude and power spectral densities.

to you. These functions provide Qucs with powerful numerical resource which significantly extends the range of problems that Qucs can analyse.

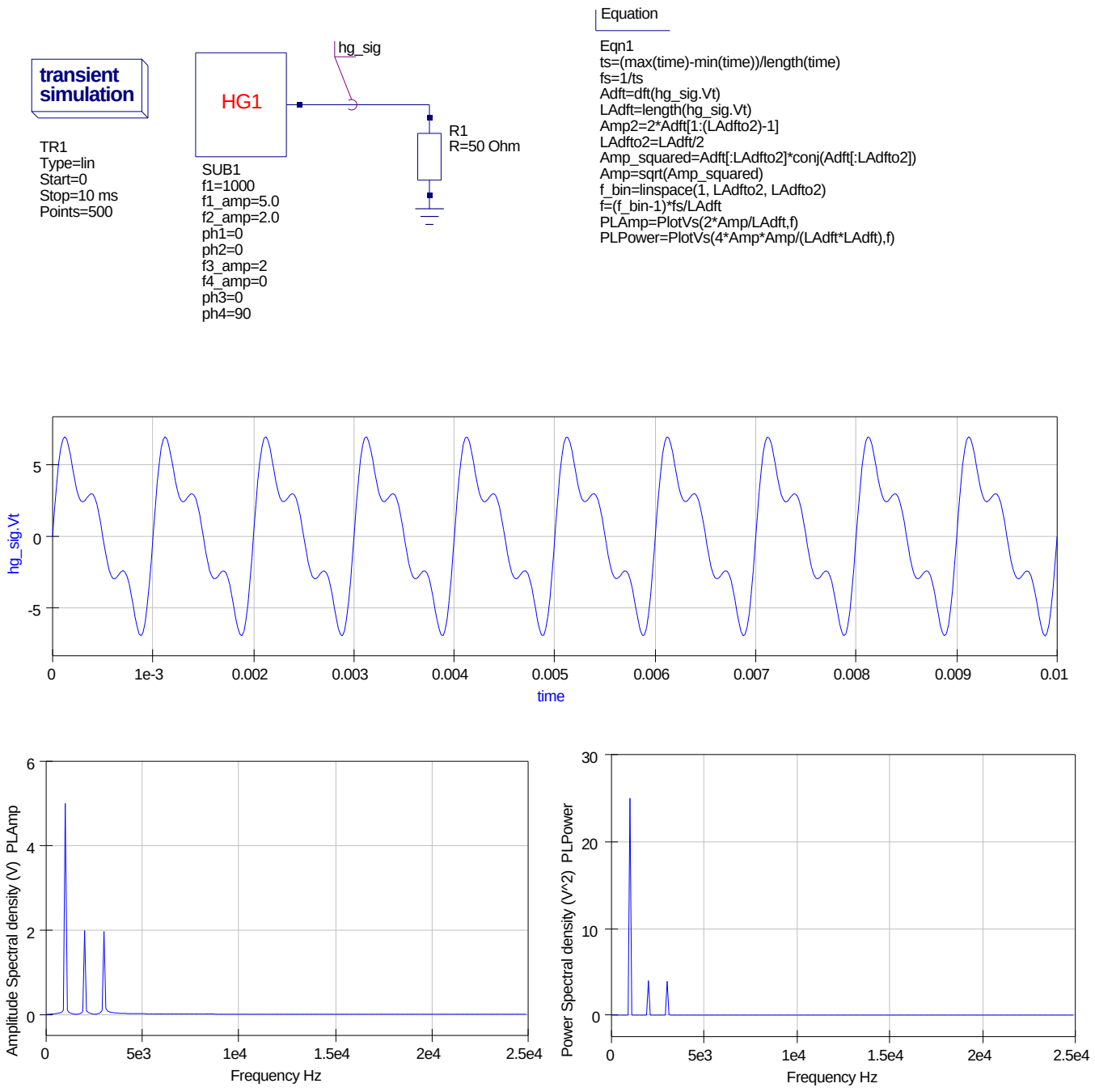


Figure 16.8: Harmonic generator subcircuit test circuit and simulation waveforms

16.5 Building universal macromodels using subcircuits and parameters

Passing parameters to subcircuits allows universal macromodels to be built. One obvious application of this technique is the modelling of operational amplifiers (OP AMP) and other integrated circuits. The approach adopted is similar to that outlined in the last section. However, because of the complexity of the models it is advisable to break a model into a series of smaller blocks. These are then combined to form a complete subcircuit macromodel. Two techniques are possible when partitioning models, these are demonstrated next. Shown in Fig. 16.9 is a simple AC OP AMP model[21] consisting of an input stage, an intermediate gain stage and an output stage[22]. An equation block, if needed, is associated with each stage. These blocks contain the equations for calculating the component values in a given stage. A single schematic symbol represents the model. This has a list of parameters attached. The flow of information into a macromodel starts with parameters passed into a subcircuit, via a schematic symbol, then onto the equation blocks, where it is finally used to calculate the component values. Hence, by simply changing the subcircuit parameters different OP AMPs can be simulated using a single generalised macromodel. However, please note that different OP AMP circuit structures, or indeed technologies, naturally result in a series of generalised subcircuit macromodels to cover all possible types in a given device family. The second technique involves breaking a model down into smaller blocks and associating subcircuit symbols with each block. This approach is illustrated in Fig. 16.10. Again parameters are passed from the top level symbol (called AC in the schematic) to the inner subcircuits. These pass their own parameters down a subcircuit level where the component calculations are completed. The second technique results in two levels of subcircuit, accounting for the change in parameter name when passing a parameter from top to lower hierarchy. A second more detailed example showing how to construct nested subcircuits is presented later in these notes.

In reality the macromodel for a typical OP AMP that models DC, AC and transient domains is much more complex than the model given in Fig. 16.9. The schematic for a typical multi-domain OP AMP modular macromodel is shown in Fig. 16.11, where each section of the macromodel is represented, if needed, by it's own equation block.

The test schematics shown in Figures. 16.12 and 16.13 show two OP AMPs with different subcircuit parameters. In Fig. 16.12 the small signal characteristics of unity gain closed

[21] The term AC here refers to the fact that the OP AMP model chosen for demonstration purposes is a simplified version of a multi-domain OP AMP model. It only models small signal AC parameters and device input stage bias and offset properties.

[22] The schematic shown in Fig. 16.9 forms part of a modular OP AMP macromodel. A detailed description of the function of individual networks and the derivation of the component equations is given in Qucs tutorial Modelling Operational Amplifiers, Mike Brinson, `http://qucs.sourceforge.net/docs.html`.

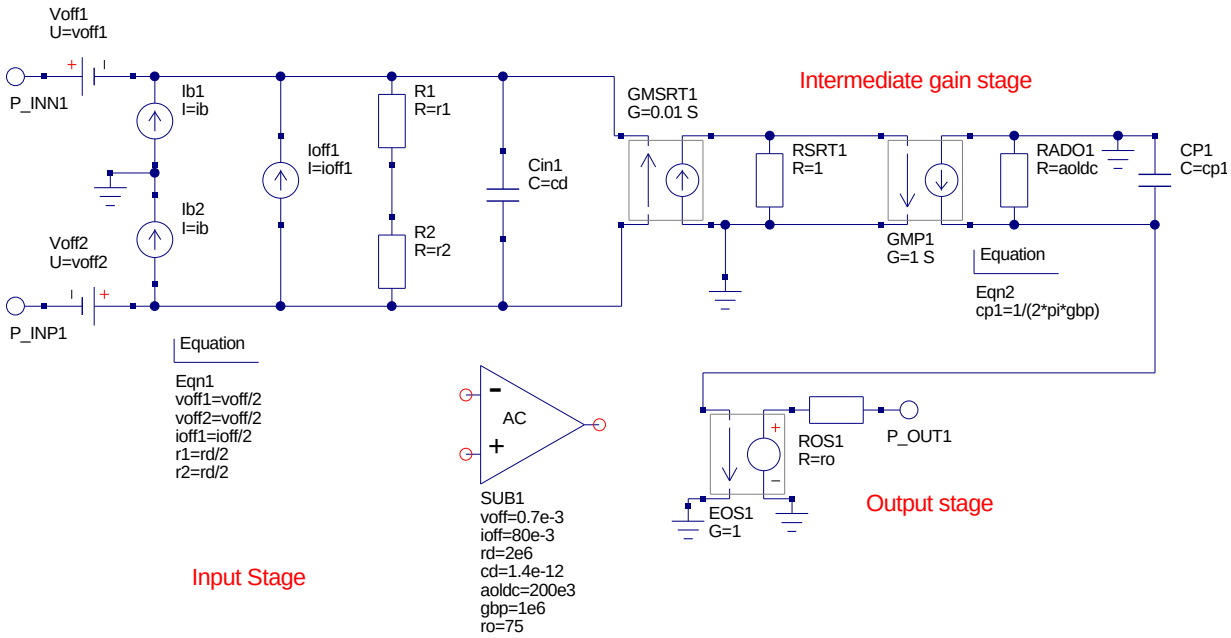


Figure 16.9: Expanded AC OP AMP model showing circuitry and equation blocks

loop amplifiers clearly show the difference in performance of the OP AMPs. Figure 16.13 is particularly interesting in that it illustrates how Qucs can be used to determine the effect of amplifier offset voltage on integrator DC saturation by stepping resister *rp* through a series of values. The low offset voltage of the OP27 makes this device much more suitable for integrator circuits when compared to the popular UA741. These results can be confirmed by a simple calculation: the offset voltage for the UA741 is set at 0.7 mV and the amplifier open loop DC gain at roughly $200,000$. The UA741 goes into saturation when *rp* is approximately 20 MΩ. In saturation the OP AMP gain becomes open loop giving a DC output voltage of roughly $0.7\text{e-}3 \cdot 2\text{e}5$ or 14 V, which agrees with the Qucs simulation results.

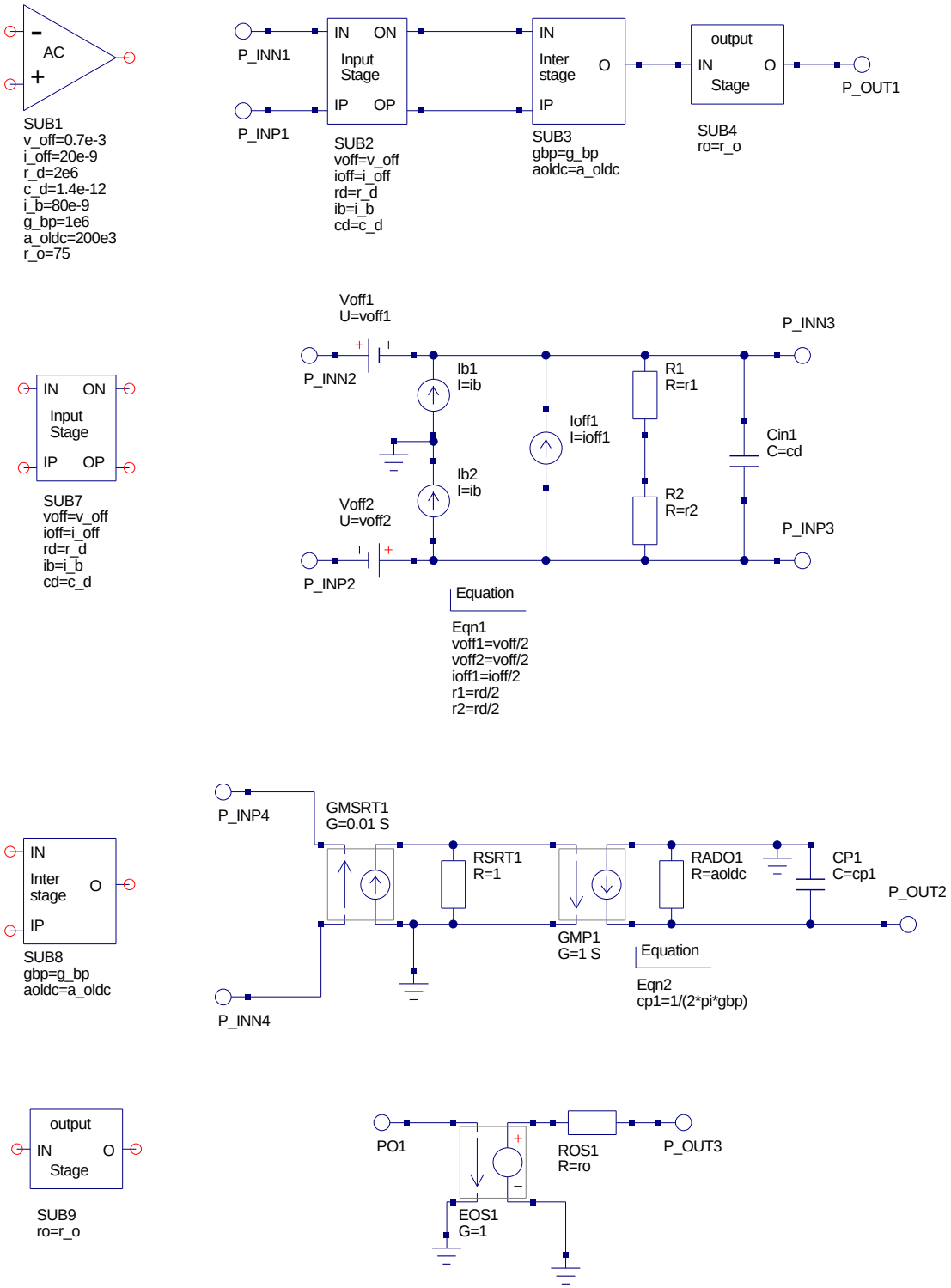


Figure 16.10: Modular AC OP AMP model showing subcircuits

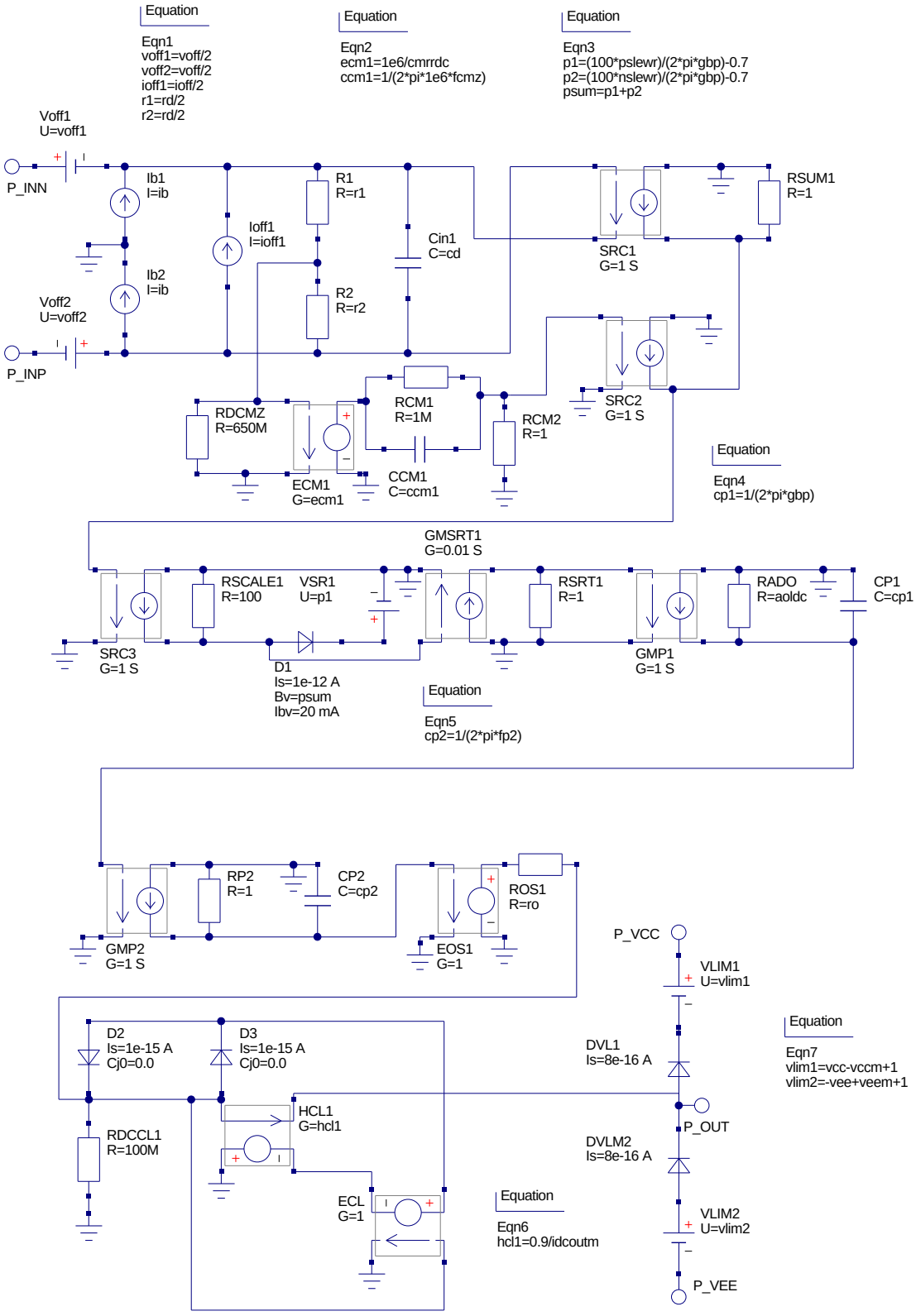


Figure 16.11: Modular OP AMP subcircuit schematic with embedded component calculation equations

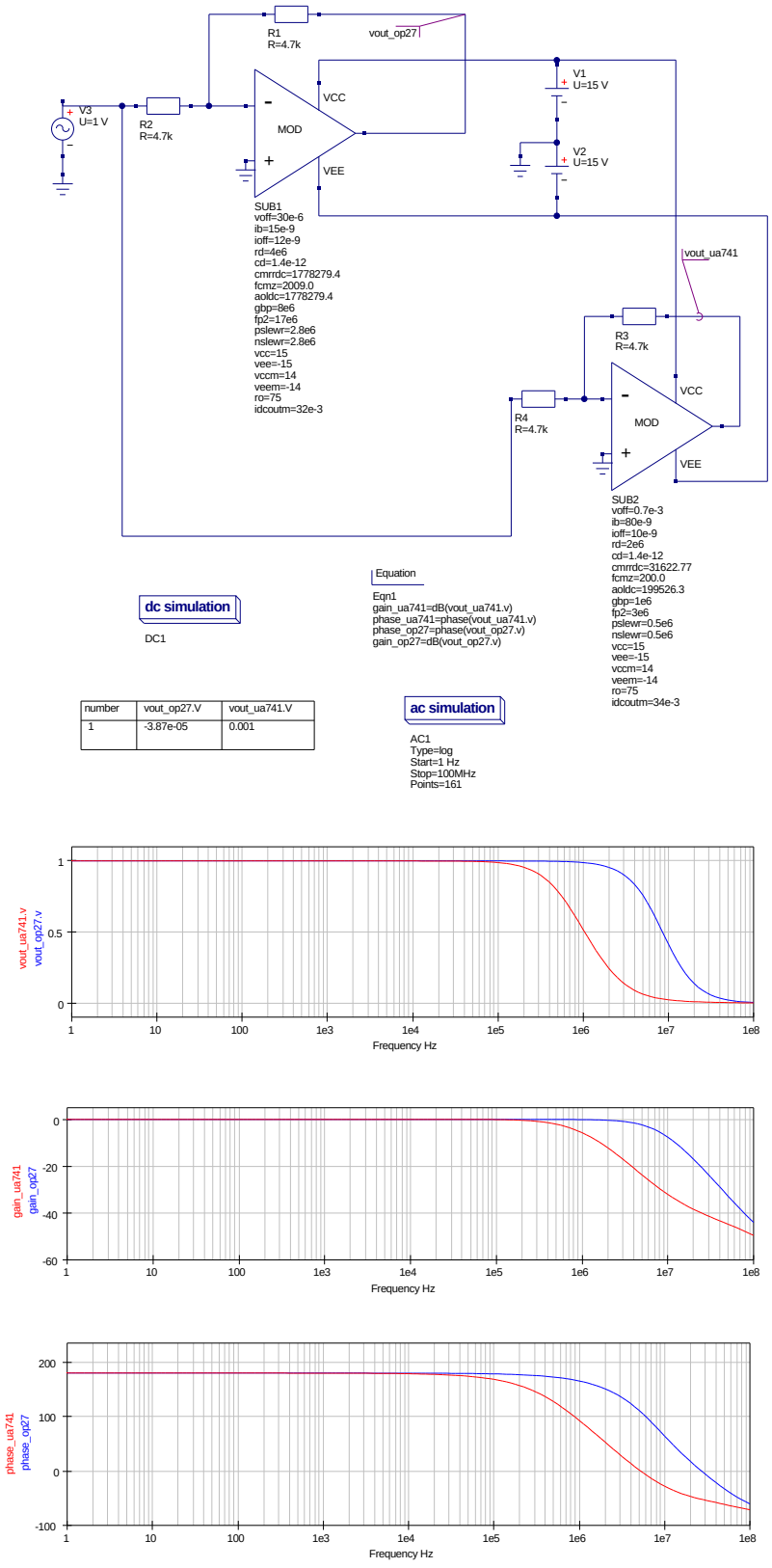
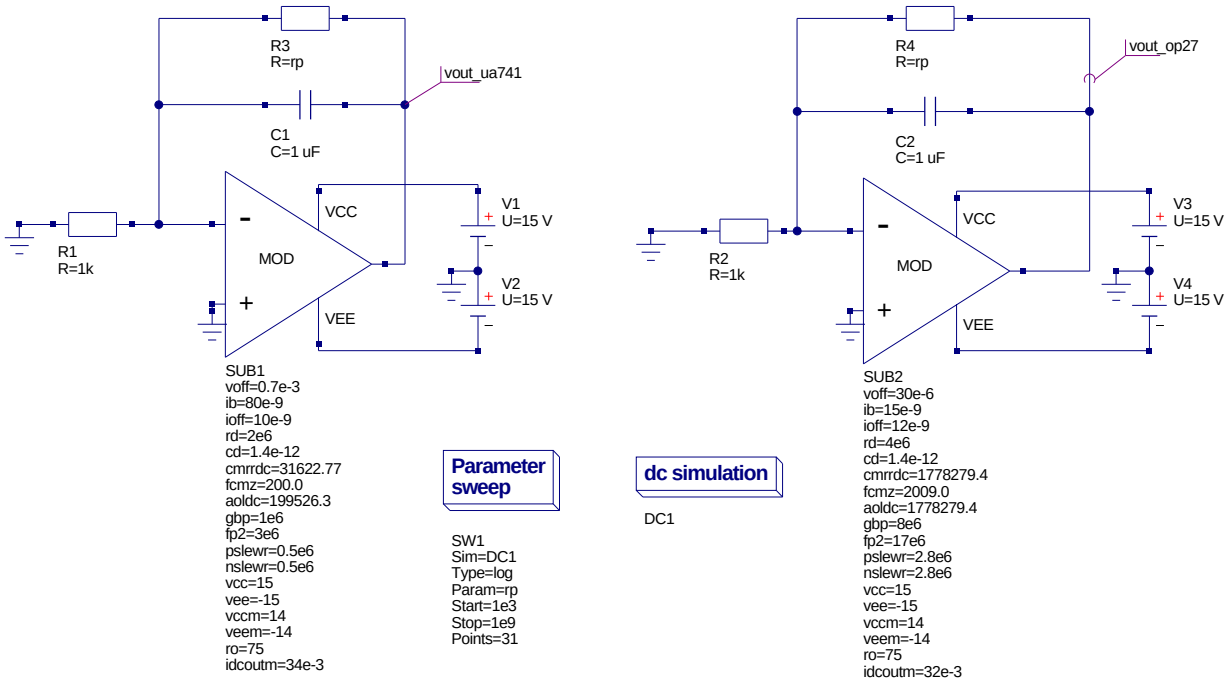


Figure 16.12: Unity gain OP AMP test circuit and waveforms



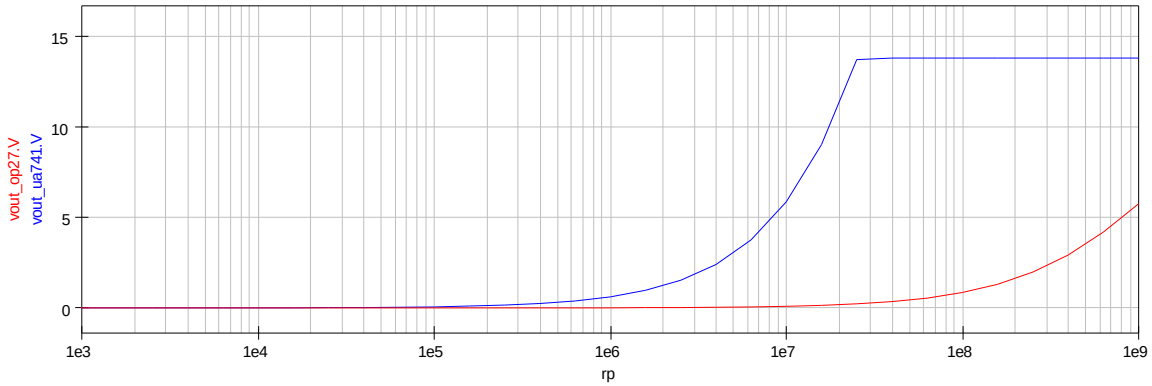


Figure 16.13: Integrator test circuits for determining DC saturation

16.6 More complex nested subcircuit models

In the previous two sections the example circuits only included subcircuits nested to one or two levels. Qucs does however, allow subcircuits to be nested to an arbitrary level and parameters can be passed down the nested chain to any depth required. Some care is needed when setting up the parameter passing sequence. Shown in Fig. 16.14 is a top level subcircuit with temperature swept between 10 and 110 centigrade. A simple resistor voltage divider network is at the bottom of a series of linked subcircuits, three levels down. R2 in the divider is a function of temperature. A schematic representation of the coupled subcircuits parameter passing sequence is also given in the right hand side of Fig. 16.14. Each level passes the value of temperature to it's next lower member in the hierarchy. The Qucs generated netlist given in Fig. 16.15 clearly shows the parameter passing mechanism employed by Qucs. The ability to nest subcircuits and pass parameters down a hierarchy is an important feature in Qucs because it allows both circuit design and device data to be passed to different sections of the circuit/system being simulated. These parameters can, of course, be at different levels in a problem hierarchy providing a very flexible and powerful design/analysis tool.

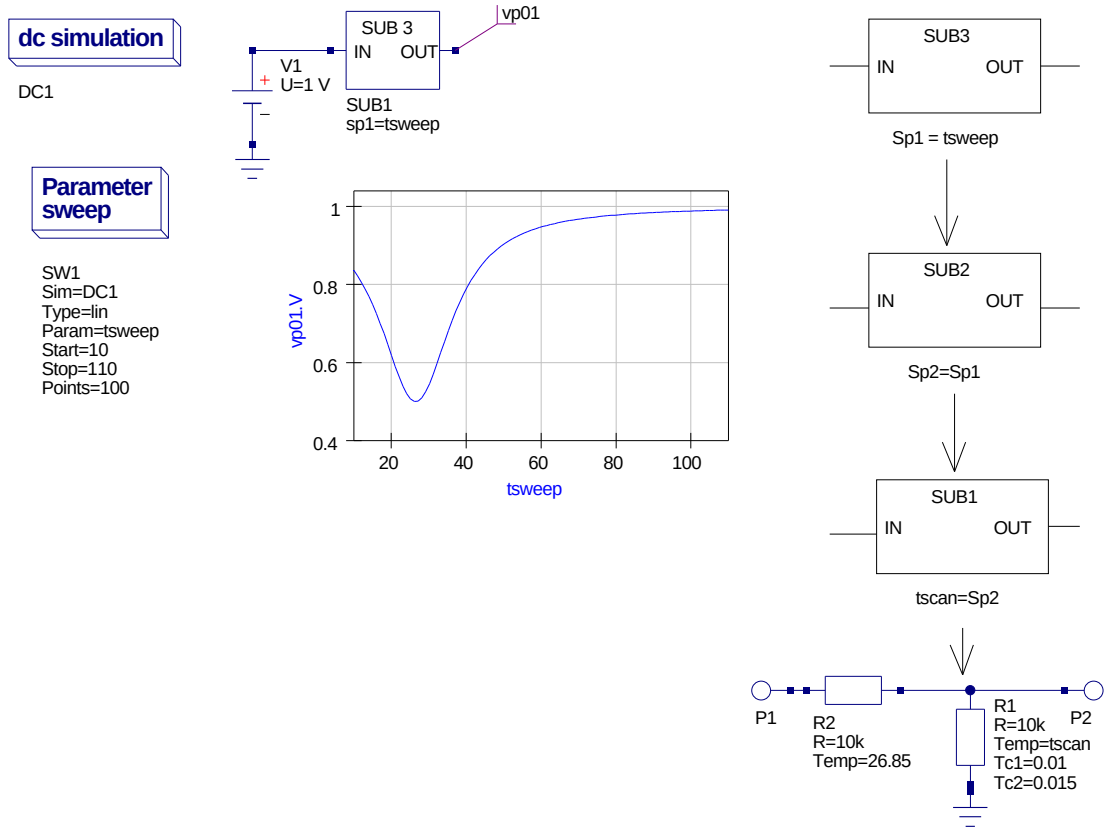


Figure 16.14: A nested subcircuit showing parameter passing sequence

```
# Qucs 0.0.12  /media/hda2/Qucs_equation_modelling_prj/rdiv_test_tsweep_3l.sch
.Def:rdiv_sub1_temp _net1 _net0 tscan="27"
R:R2 gnd _net0 R="10k" Temp="tscan" Tc1="0.01" Tc2="0.015" Tnom="26.85"
R:R1 _net1 _net0 R="10k" Temp="26.85" Tc1="0.0" Tc2="0.0" Tnom="26.85"
.Def:End

.Def:rdiv_test6_temp _net1 _net0 sp2="27"
Sub:SUB1 _net1 _net0 Type="rdiv_sub1_temp" tscan="sp2"
.Def:End

.Def:rdiv_sub3_temp _net0 _net1 sp1="27"
Sub:SUB1 _net0 _net1 Type="rdiv_test6_temp" sp2="sp1"
.Def:End

Vdc:V1 _net0 gnd U="1 V"
.DC:DC1 Temp="26.85" reltol="0.001" abstol="1 pA" vntol="1 uV"
saveOPs="no" MaxIter="150" saveAll="no" convHelper="none" Solver="CroutLU"
.SW:SW1 Sim="DC1" Type="lin" Param="tsweep" Start="10" Stop="110" Points="100"
Sub:SUB1 _net0 vp01 Type="rdiv_sub3_temp" sp1="tsweep"
```

Figure 16.15: Qucs netlist for nested subcircuit showing parameter passing sequence

16.7 Introduction to equation defined devices (EDD)

Although adding symbolic equations to a simulator merges circuit design and analysis, it is by making these equations functions of circuit variables that the real power of modern circuit simulator is fully exploited. Equations that are functions of voltage, current and charge have to be continuously evaluated as a simulation progresses. This is in contrast to the type of equations previously introduced, which are only evaluated at the start of a simulation sequence. When component properties are functions of circuit variables considerable complexity is added to a simulation engine and as a result most simulators restrict such properties to a small number of component types, the most common being controlled current and voltage generators[23]. Qucs version 0.0.12 introduces an equation defined device (EDD) which allows it's terminal currents to be functions of voltage, and it's stored charge to be functions of voltage and current. The EDD is similar, but more advanced, to the B type controlled source implemented in SPICE 3f5. It is capable of realising the same models as the SPICE B type device plus an extensive range of more complex compact device models. At this stage in Qucs development only the explicit

[23]Probably the most well known non-linear controlled generators are the SPICE 2g6 and 3f5 forms, see A. Vladimirescu, Kaihe Zhang, A.R. Newton, D.O. Pederson and A. Sangiovanni-Vincentelli, SPICE Version 2G User's Guide, 1981, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Ca. 94720, section 11, Appendix B: Nonlinear dependent sources., and B. Johnson, T. Quarles, A.R. Newton, D.O. Pederson and A. Sangiovanni-Vincentelli, SPICE3 Version f User's Manual, 1992, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Ca. 94720, section 3.2.2.4, Non-linear dependent sources.

form of EDD is implemented[24]. EDD is an advanced component that allows Qucs users to construct their own device models from a set of equations derived from the physical properties that characterise a device. The explicit form of EDD can only be used to develop models for devices where their defining equations can be transformed into the explicit analysis form required by Qucs[25]. A range of functions similar to those defined in the Verilog-A compact device modelling language are provided by Qucs, making the equation modelling language easy to use and powerful. The ternary ? : form of the C language if statement has also been implemented to allow selection of model equations that change with differing device voltage, current and charge conditions. Before introducing the EDD symbol and it's properties consider the following circuit simulation modelling problem: a model for a device is required where the output voltage is a function of two input voltages VIN_1 and VIN_2, such that

$$V_{out}\left(VIN_1, VIN_2\right) = VIN_1 \cdot VIN_2, \tag{16.11}$$

where VIN_1 and VIN_2 can be arbitrary varying voltages.

This type of model is difficult to simulate at functional level[26] using the pre-version 0.0.12 built-in devices. A linear voltage controlled voltage source can be used to multiply a voltage by a constant. Multiplying by a second voltage is not possible with the linear controlled sources. Qucs AM modulated and PM modulated sources are the nearest that Qucs has to the source defined above. These sources however, only allow sinusoidal carrier signals. Illustrated in Fig. 16.16 is a four quadrant multiplier EDD which allows multiplication of two varying signals[27]. The EDD device generates current $I1 = V2 \cdot V3$. This in turn is transformed to the output voltage by a unity gain current controlled voltage source SRC1. An EDD device can consist of up to 8 branches. The branches have currents, I1 to I8, voltages V1 to V8 and internal charges Q1 to Q8 respectively. Overall the total device current depends how these branches are connected. A similar comment applies to the total device charge. In Fig. 16.16 currents I2 and I3 are set to zero, charges Q2 and Q3 are also zero, and voltages $V2 = VIN_1$ and $V3 = VIN_2$. Hence current I1 becomes the multiplication of VIN_1 and VIN_2. The fact that currents I2 and I3 are set to zero implies that the terminals connected to the external input voltages have high impedance and act as voltage probes. The test circuit in Fig. 16.16 is shown with signal inputs generated by sinusoidal oscillators; V1 acts as a modulating signal and V2 as a carrier signal. The bottom right hand corner of Fig. 16.16 includes a second graph which illustrates the effect

[24]See Qucs Technical Papers, Section 10.7: Equation defined models, Stefan Jahn, Michael Margraf, Vincent Habchi and Raimund Jacob, `http://qucs.sourceforge.net/technical.html`.

[25]The Y parameters of the device being modelled must also exist for the explicit form of the EDD to be valid.

[26]It is, of course, possible to model the multiplier operation at discrete component level e.g. using a Gilbert cell mixer circuit.

[27]This model is based on an idea suggested by Stefan Jahn, during the EDD development phase.

of changing signal V2 to a square wave source with 0.05ms period.

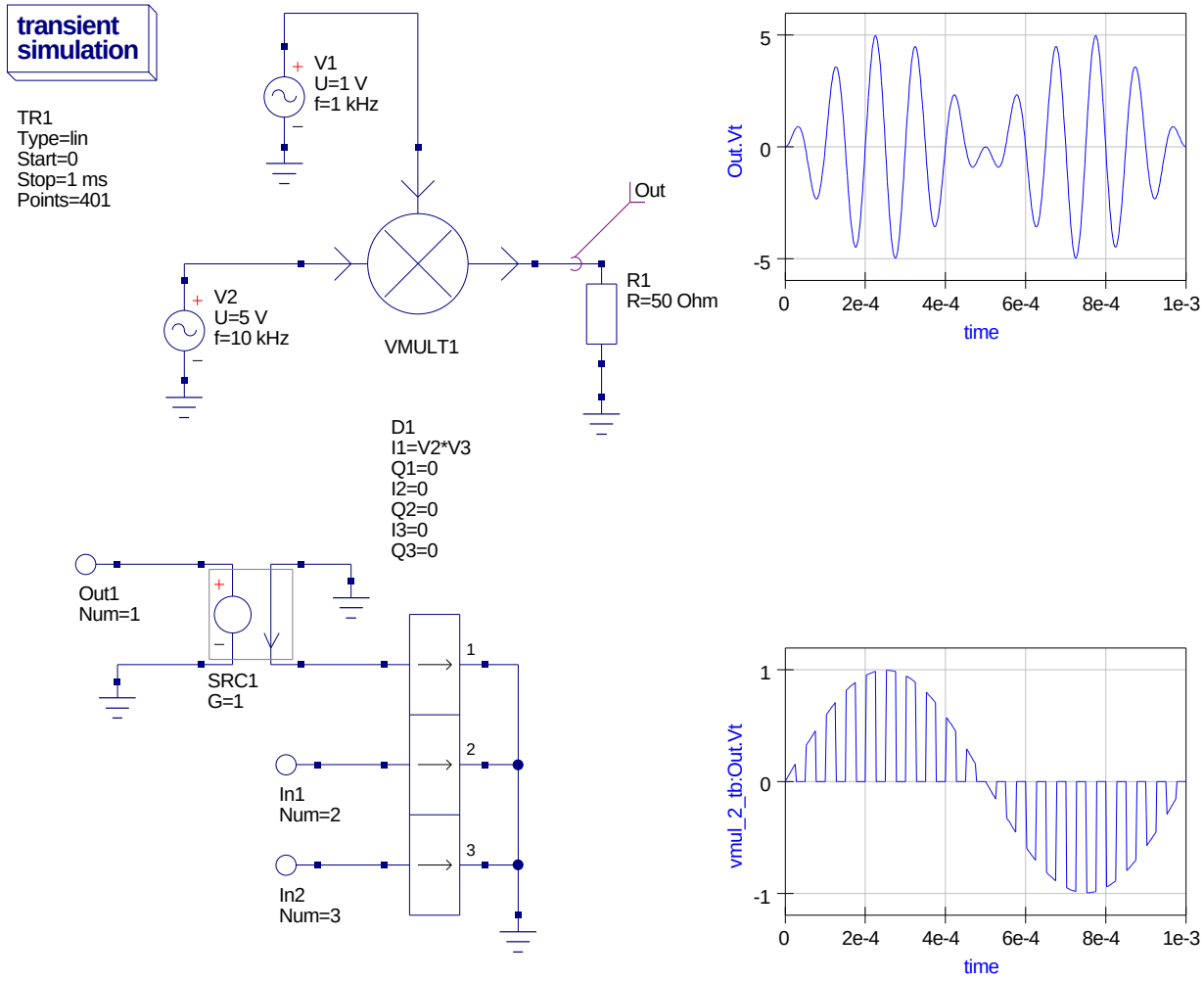


Figure 16.16: Qucs EDD four quadrent multiplier model and test circuit

16.8 The Qucs EDD component

A two terminal model for a universal non-linear component with resistive, capacitive and inductive parallel branches is shown in Fig. 16.17. All three branches have elements that can be functions of either voltage or current or charge[28]. The Qucs EDD component can be used to model this nonlinear device. One EDD element is needed to model the resistive and capacitive branches. A second EDD device, plus a gyrator, models the inductive branch.

[28]Each branch can be a function of one or more of these circuit variables but not necessarily all three at the same time.

The total terminal current is the sum of the individual branch currents. Equations for the three branch currents are given by the following equations:

$$I = I1 + IC + IL, \tag{16.12}$$

where

$$I1 = f(V), \;\; IC = C(V,I) \cdot \frac{dV1}{dt} = \frac{dQ1}{dt} \tag{16.13}$$

Also

$$V1 = i2, \;\; V2 = -IL, \;\; i2 = -L(I) \cdot \frac{dV2}{dt}, \;\; V1 = L(I) \cdot \frac{dIL}{dt} \tag{16.14}$$

Giving

$$IL = \frac{1}{L(I)} \cdot \int V2 \cdot dt \tag{16.15}$$

and

$$VL = V2 = V1 = \frac{d\Phi}{dt} \tag{16.16}$$

Hence

$$I = f(V) + C(V,I) \cdot \frac{dV1}{dt} + \frac{1}{L(I)} \cdot \int V1 \cdot dt \tag{16.17}$$

The EDD is characterised by eight parallel branches each comprising a current component In and a charge component Qn, where n ranges from 1 to 8. The currents may be constants or defined by equations that are functions of the EDD branch voltages (these are designated $V1$ to $V8$). This form of the EDD component is known as the explicit EDD model. Please note, EDD currents cannot be functions of current. However, with release 0.0.12 implementation of the explicit EDD the device charge can be a function of either voltage or current[29]. The current in the resistive branch being a function of EDD voltage allows a range of two terminal[30] devices to be modelled, allowing, for example, nonlinear resistors and diode models to be easily developed. Similarly, the fact that the EDD charge can be a function of voltage or current extends the range of allowed Qucs capacitor types opening new areas of application. The same comments apply to the nonlinear inductors where components that have inductance values which are functions of current allow modelling of nonlinear transformer and coupled inductor effects. This was not possible with earlier Qucs releases. The EDD current and charge values may be defined by symbolic equations that include the operators and functions listed in the "Short description of mathematical functions" entry in the Qucs help index[31].

[29]This allows modelling of semiconductor capacitive effects where the amount of stored charge is either a function of voltage (depletion layer capacitance), or a function of current (diffusion capacitance).

[30]The number of device terminals can be increased to model transistors and other devices.

[31]The Qucs operators and functions are a superset of those defined in the Verilog-A language manual. However, in some cases the name of the operator or function differs slightly. For example Verilog-A uses $pow(x,y)$ for the power function whilst Qucs uses $\wedge$ to denote x^y. An example of differing function names are the inverse trigonometric functions. A list of the available functions is given in Appendix A.

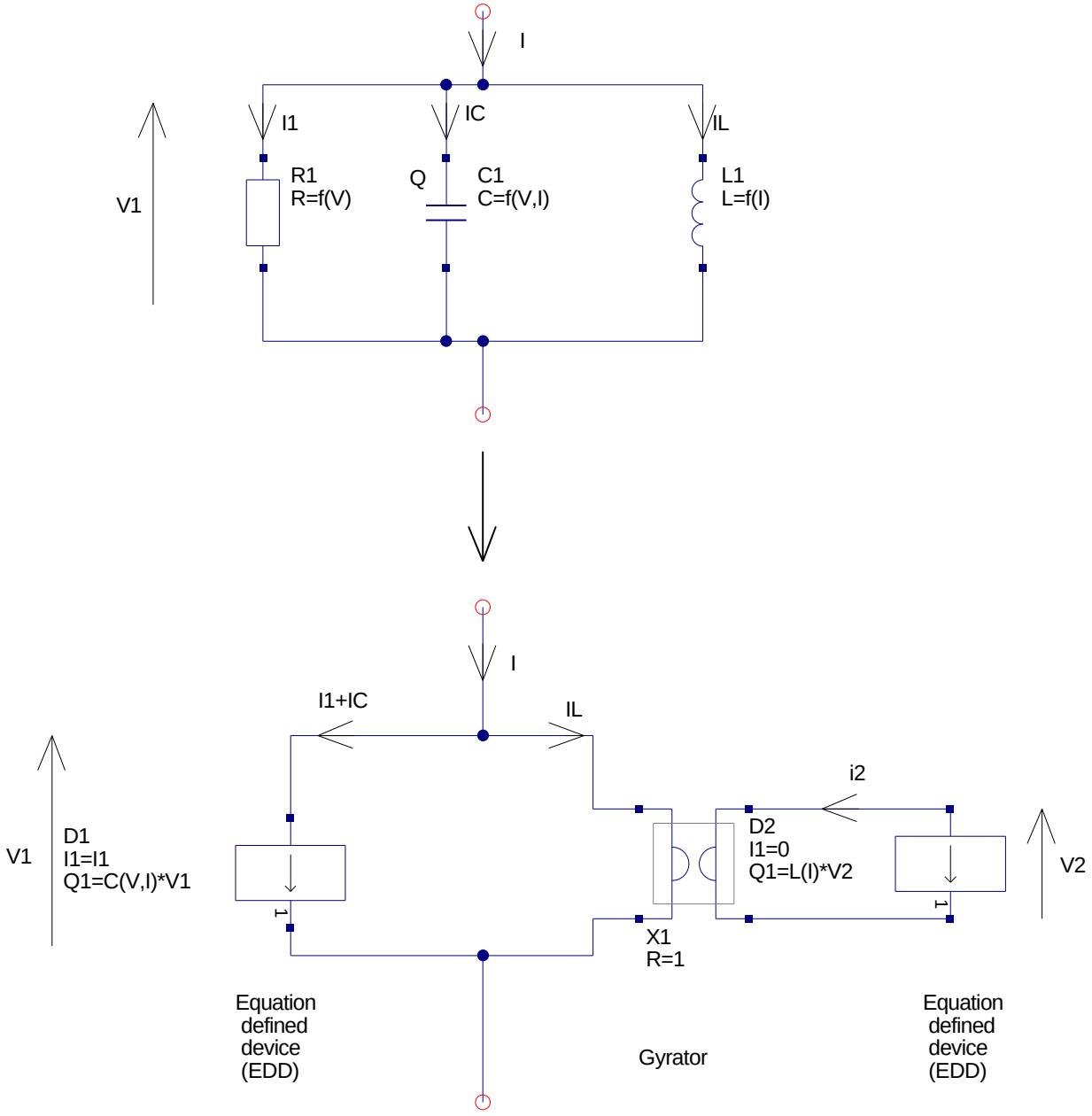


Figure 16.17: A non-linear two terminal branch with parallel resistive, capacitive and inductive components

16.9 Modelling nonlinear resistors

In many measurement applications a transducer is employed to transform changing values of a physical quantity to, say, changes in resistance. Often the resistive characterstics of these devices are nonlinear. To demonstrate how the EDD can be used to model a nonlinear resistance the example shown in Fig. 16.18 is introduced. In this schematic an EDD represents a resistance that is a function of the applied voltage across it's terminals. This example deliberately shows an extreme case where the resistance changes in a resistive pulse like fashion as the terminal voltage increases. The example also introduces for the first time the ternary ? : operator and illustrates how it can be nested to give an "if then else" structure to define the component properties. A point of note with these very nonlinear devices centres around the fact that it is possible to define components that have discontinuities in their I-V characteristics[32]. The EDD current equation defines how the resistance of this device changes with changing terminal voltage. This equation is given by

```
I1=V1/((V1<1.0) ? 1000 : (V1<2.0)
                ? 1000+4000*(V1-1) : (V1<5.0)
                ? 5000 : ((V1 >=5.0) &&  (V1<6.0))
                ? 5000-4500*(V1-5.0) : 500)
```

Which in terms of an "if then else" type statement is equivalent to:

```
I1 = V1/( if (V1 < 1.0) then 1000
          else if (V1 <  2.0) then 1000 + 4000*(V1-1)
          else if (V1 <  5.0) then 5000
          else if ((V1 >= 5.0) && (V1 < 6.0)) then 5000 - 4500*(V1-5.0)
          else 500 )
```

[32]One effect of such a discontinuity is the introduction of rapidly changing circuit conditions which can cause the simulator difficulties in converging to a correct solution. Sometimes, if this happens, simulation run times may be dramatically increased or simulation fails altogether.

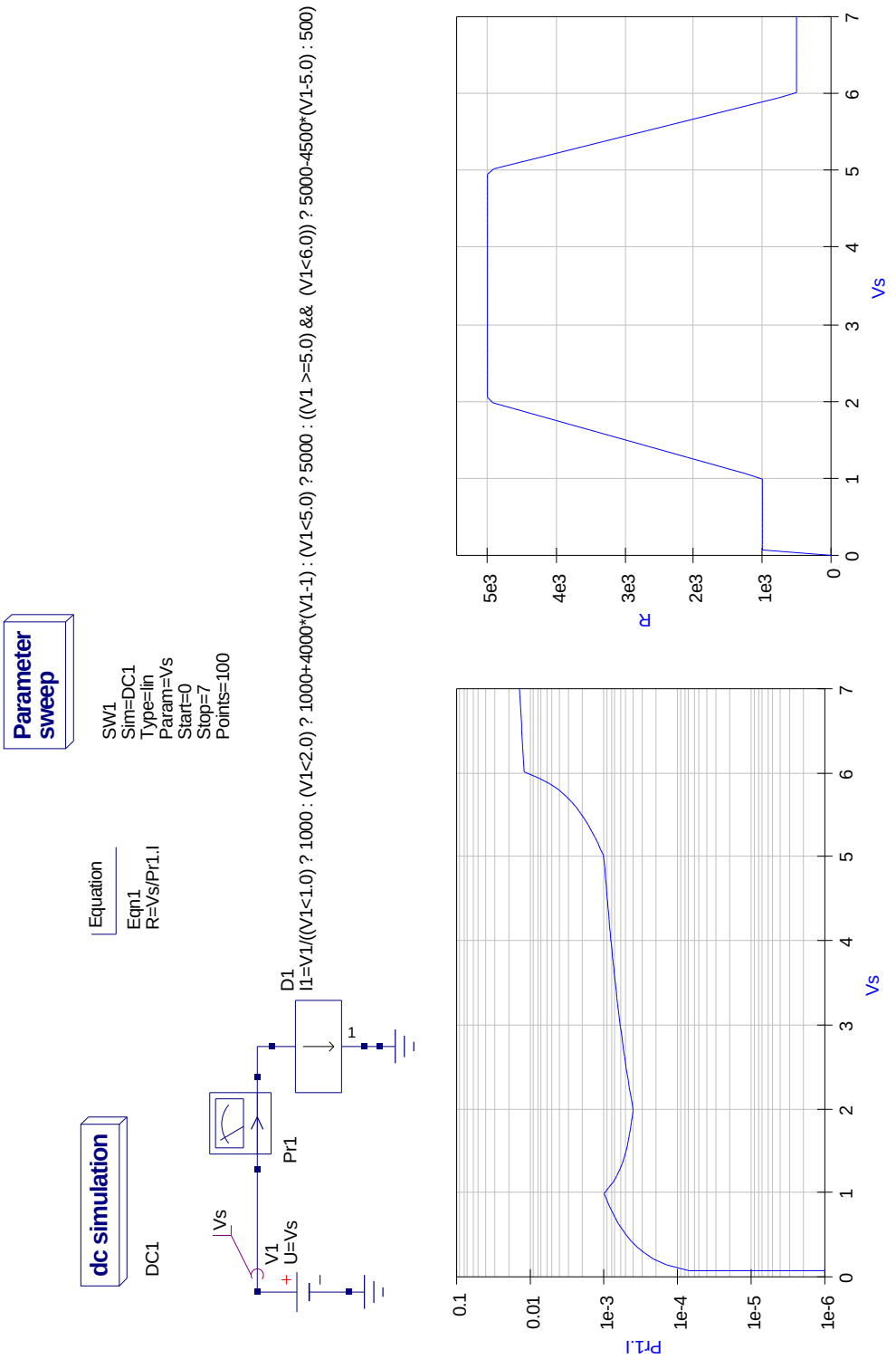


Figure 16.18: Qucs nonlinear resistor model

16.10 Modelling nonlinear capacitors and inductors

Nonlinear capacitors, who's C value is a function of terminal voltage, and nonlinear inductors, who's L value is a function of terminal current, commonly act as control elements in electronic systems. SPICE 2g6 includes a nonlinear symbolic polynomial form of C and L[33]. The schematic shown in Fig. 16.19 illustrates how a nonlinear capacitor can be modelled by an EDD. This model is based on a SPICE like polynomial function with four coefficients; C0, C1, C2 and C3[34]. The test circuit is a simple RC network with nominally identical R and C component values to those shown in Fig. 16.2. Increasing the value of DC source V1 also increases C which in turn decreases the RC low pass filter -3dB frequency. This effect is very visible in Fig. 16.19. The nonlinear changes in C are also clearly illustrated in the output voltage and phase curves. The schematic symbol for the nonlinear capacitor is shown in Fig. 16.19 with a red ring drawn around the normal capacitor symbol. This denotes an EDD based component. An alternative convention is to use red lettering within a symbol. The test circuit and simulation results for a nonlinear inductance are shown in Fig. 16.20. The EDD model is similar to the SPICE 2g6 nonlinear inductance model with four coefficients. This number can be increased, if required, by extending the EDD polynomial expression. A gyrator is employed with the EDD to model the nonlinear inductance. The effect of nonlinear inductance on the inductance current is shown by the difference between probe currents *Pr1* and *Pr2*.

[33]The details of these polynomial functions are presented in Test Reports 4 and 5 of the SPICE to Qucs testing Series, Mike Brinson, `http://qucs.sourceforge.net/docs.html`.

[34]SPICE 2g6 allows up to twenty coefficients. Simply add more higher order terms to the Qucs polynomial if required.

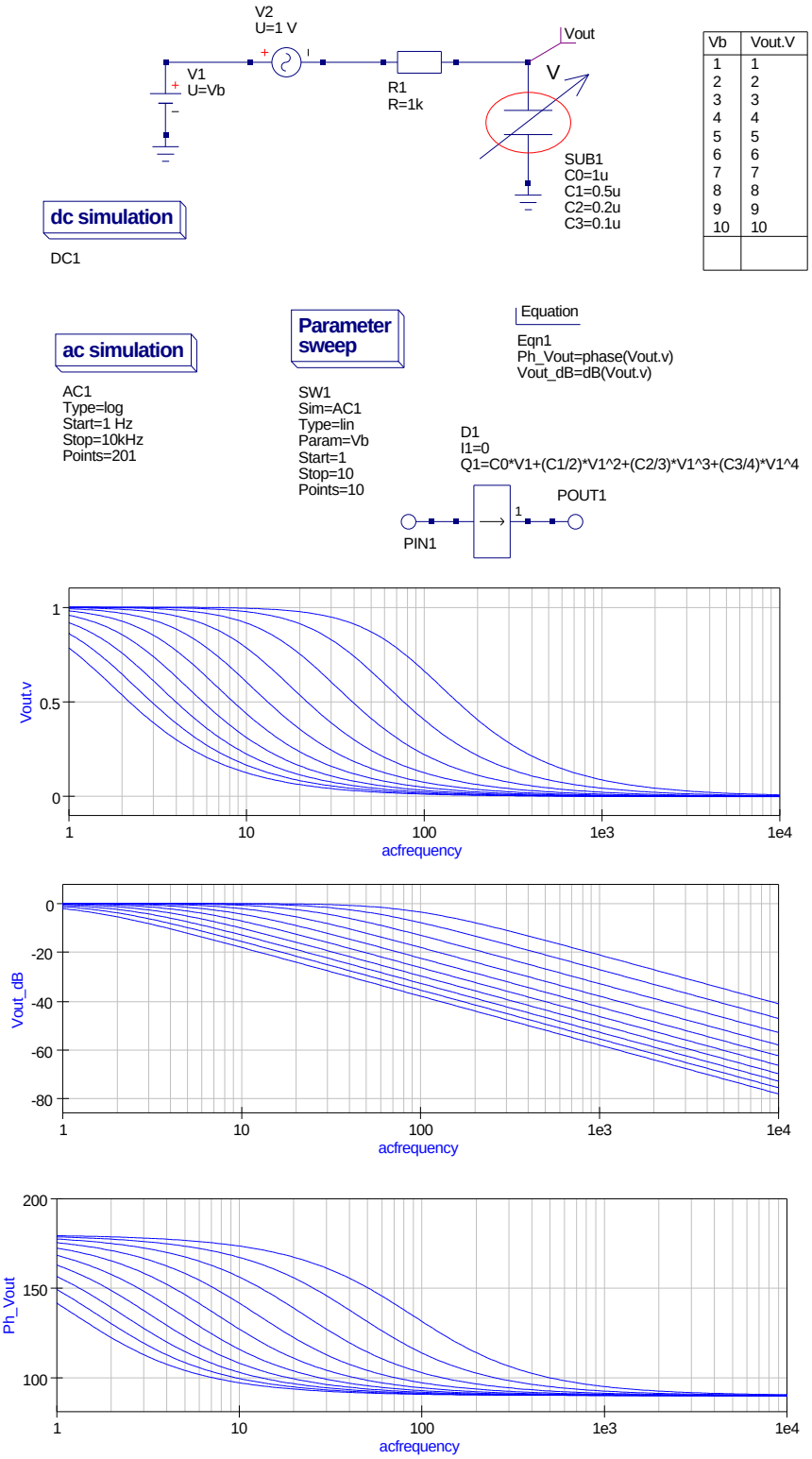


Figure 16.19: Qucs nonlinear capacitor model

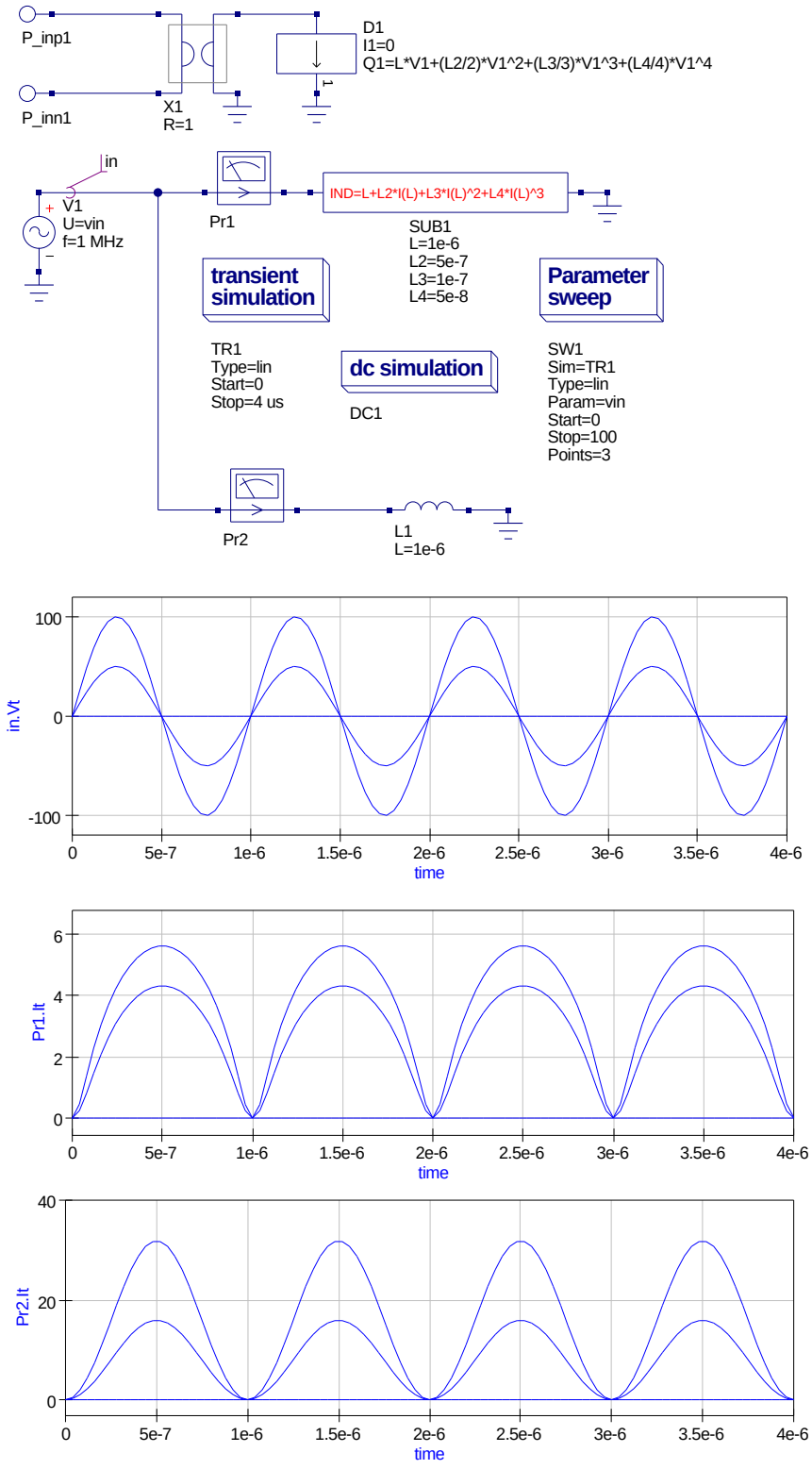


Figure 16.20: Qucs nonlinear inductor model

16.11 Compact device modelling using EDD

Semiconductor device models are a corner stone of all circuit simulators. Often they are characterised by the same parameters as those found in the SPICE 2g6 and 3f5 diode, BJT, FET and MOS models.[35]. Since the original SPICE semiconductor device models where first developed many new extensions to these models have been proposed. Unfortunately, adding such models to a circuit simulator is a complex process, being both time consuming and requiring specialised knowledge. For the average Qucs user the hand coded C++ model generation route is one that they would not contemplate attempting because of the depth of knowledge and specialised skills required. The Qucs EDD was devised to promote fast, and straight forward, prototyping of semiconductor compact models, allowing a wider Qucs population the opportunity to try their hand at device model construction. To demonstrate the stages needed to generate an EDD model of a semiconductor device a compact model of a diode is introduced in this section[36].

The DC diode current I_d is given by the following functions of diode voltage V_d[37].

$$I_d = I_s \cdot \left(\exp\left(V_d/(n \cdot Vt) - 1\right) + V_d \cdot GMIN, \quad \forall \left(-5 \cdot n \cdot Vt \leq V_d\right)\right. \tag{16.18}$$

$$I_d = -I_s + V_d \cdot GMIN, \quad \forall \left(-BV < V_d\right) \quad \text{and} \quad \left(V_d < -5 \cdot n \cdot Vt \leq V_d\right) \tag{16.19}$$

$$I_d = -IBV, \quad \forall \left(V_d = -BV\right) \tag{16.20}$$

$$I_d = -I_s \cdot \left(\exp\left(-(BV + V_d)/Vt\right) - 1 + BV/Vt\right), \quad \forall \left(V_d < -BV\right). \tag{16.21}$$

In these equations:

- I_s = the saturation current.
- n = the emission coefficient.

[35]The SPICE 2g6 and 3f5 device parameters are a subset of those commonly provided with current generation of circuit simulators, including Qucs.

[36]A second three terminal MESFET transistor example is available for downloading from the Qucs Web site.

[37]These equations are for the SPICE 2g6 diode model, see Giuseppe Massobrio, Chapter 1, Pn-junction diode and Schottky diode, Semiconductor device modeling with SPICE, Edited by Paolo Antognetti, Giuseppe Massobrio, 1988, McGraw-Hill,Inc, ISBN 0-07-002107-4.

- $GMIN$ = a small conductance in parallel with the diode[38]
- $Vt = kB \cdot T/q$, where T is the diode temperature in Kelvin, kB is Boltzmann's constant and q the charge on the electron.
- BV = reverse breakdown voltage (positive number)
- IBV = reverse breakdown current (positive number).

Figure 16.21 gives the EDD model for the experimental semiconductor diode. The ternary operator ?: is used to select the correct equation for each diode operating region. The diode current Id : $content.tex, v1.22007/06/0316$: 58 : $59elaExp$ is the sum of EDD branch currents $I1$ to $I4$, where $I1$ represents the diode forward bias region, $I2$ the reverse bias region and $I3$ plus $I4$ the diode reverse bias breakdown region. When calculating diode current a special form of the exponential function exp(), called limexp(), is employed to assist Qucs to converge to a solution during DC and transient large signal analysis. The function limexp() linearises the exponential function at large argument values minimising the possibility of floating point overflow and generation of software exceptions. The $I_d - V_d$ characteristic curves shown in Fig. 16.21 are for the forward bias region with series resistance rs set to 0.01Ω. For completeness the simulation data for the Qucs built-in diode are also given. Clearly the two sets of results are very similar. The DC simulation results for the diode reverse breakdown region of operation are shown in Fig. 16.22. Again for comparison an $I_d - V_d$ plot for the Qucs built-in diode is also provided. In this region of operation some slight differences are apparent: although for both devices the reverse breakdown is very close to 100V the slope of the $Id - Vd$ curve at negative voltages beyond -BV is different, emphasising that the SPICE diode model does not model breakdown or zener effects well[39].

The next stage in the development of the diode model is to add capacitance effects: depletion layer capacitance for the reverse bias region and diffusion capacitance for the forward bias region. Diode capacitance is given by:

- Depletion layer capacitance

$$C_{dep} = \frac{dQ_{dep}}{dV_d} = Area \cdot Cj0 \left(1 - \frac{V_d}{V_j}\right)^{-m} \tag{16.22}$$

- Diffusion capacitance

$$C_{diff} = \frac{dQ_{diff}}{dV_d} = tt \cdot \frac{dI_d}{dV_d} \tag{16.23}$$

[38]GMIN is added to help Qucs DC convergence. The SPICE default value is 1e-12S.

[39]See Steven M. Sandler, SPICE subcircuit accurately models zener characteristics, Personal Engineering, November 1998, pp 45-48 for more information on this subject.

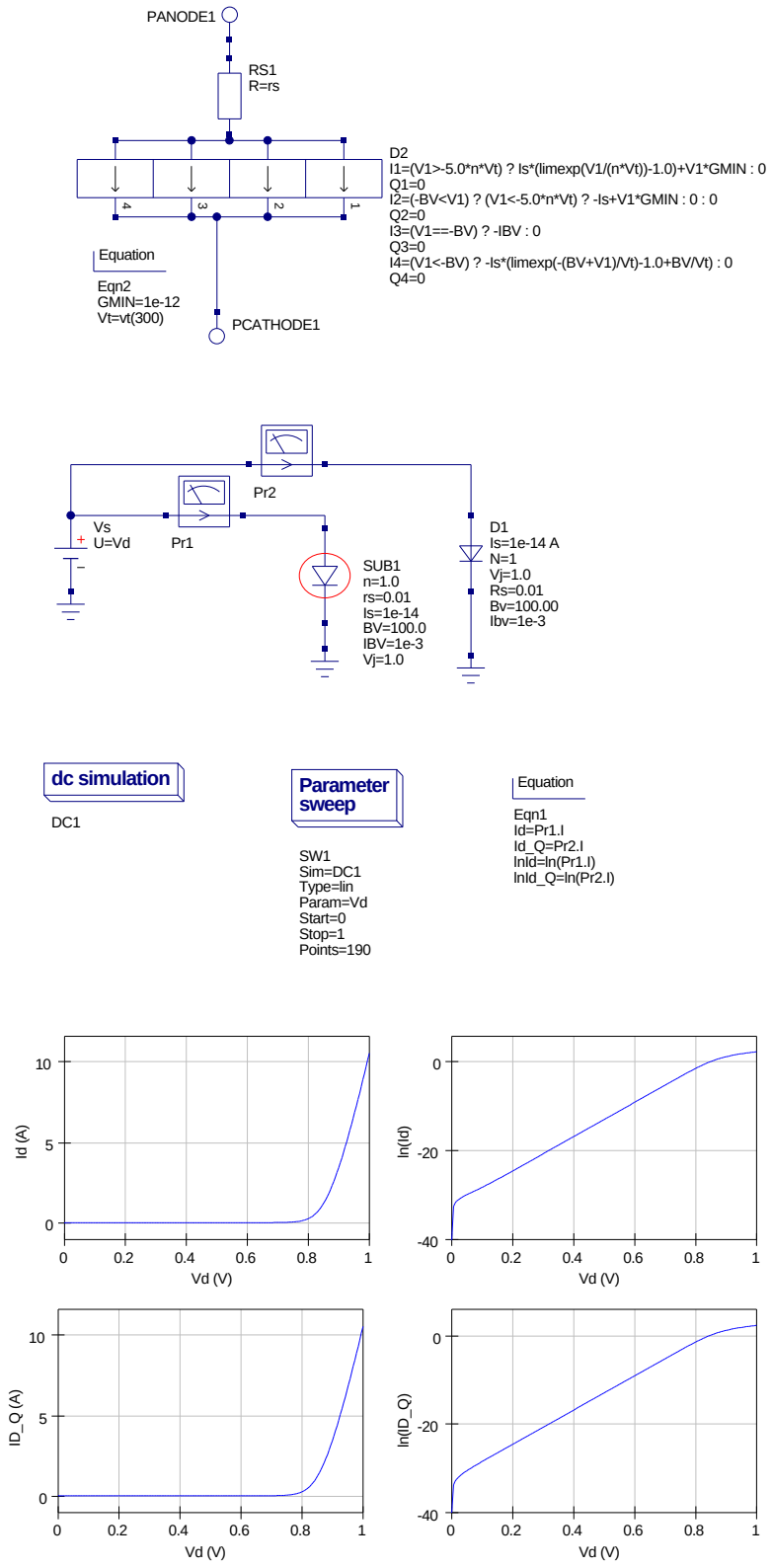


Figure 16.21: Compact diode model DC test circuit and simulation results: SUB1 is the EDD diode model and D1 the Qucs diode model with the same parameters as SUB1.

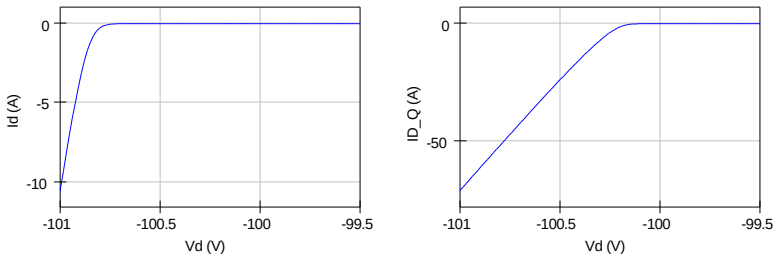


Figure 16.22: Compact diode model DC simulation results for the reverse breakdown region of operation

Where the total stored charge $Q_d = Q_{dep} + Q_{diff}$. Using the same notation as the SPICE diode model:

$$Q_{diff} = tt \cdot I_d \tag{16.24}$$

$$Q_{dep} = Area \cdot Cj0 \int_0^{V_d} \left(1 - \frac{V_d}{V_j}\right)^{-m} dV, \quad \forall \left(V_d <= FC \cdot V_j\right) \tag{16.25}$$

Using integration formula $\int (ax+b)^n dx = \dfrac{1}{a}\dfrac{(ax+b)^{1+n}}{1+n}$ and simplifying yields:

$$Q_{dep} = \frac{Area \cdot Cj0 \cdot V_j}{1-m}\left[1 - \left(1 - \frac{V_d}{V_j}\right)^{1-m}\right] \tag{16.26}$$

Also, in the forward bias region

$$Q_{dep} = Area \cdot Cj0 \cdot F1 + \frac{Area \cdot Cj0}{F2} \int_{FC \cdot V_j}^{V_d} \left(F3 + \frac{m \cdot V_d}{V_j}\right) dV, \quad \forall \left(V_d >= FC \cdot V_j\right) \tag{16.27}$$

On integrating

$$Q_{dep} = Area \cdot Cj0 \left[F1 + \left(\frac{1}{F2}\right) \cdot \left\{F3 \cdot \left(V_d - FC \cdot V_j\right) + \left(\frac{m}{2 \cdot V_j}\right) \cdot \left(V_d^2 + (FC \cdot V_j)^2\right)\right\}\right] \tag{16.28}$$

Where

$$F1 = \frac{V_j}{1-m}\left[1-(1-FC)^{1-m}\right\}, F2 = (1-FC)^{1+m}, F3 = 1-FC\cdot(1+m) \quad (16.29)$$

In these equations:

- FC = Coefficient for forward-bias depletion capacitance.
- m = Grading coefficient.
- tt = Transit time.
- $Area$ = Device area.
- $Cj0$ = Zero-bias junction capacitance.

Figure 16.23 shows the extended diode model. The C_{dep} and C_{diff} components of the device capacitance have been included in the EDD model as stored charge Q1 and Q2. Again the ternary operator ?: is employed to select the correct equation for each section of the diode DC operating range. An equation block is used to simplify the charge equations through the use of factors F1, F2 and F3.[40]. An area factor has also been added to the EDD model in Fig. 16.23. This is introduced to allow simulation of two or more equivalent parallel devices. The diode variables scaled by area are:

$$I_s(A) = I_s\cdot Area, \quad Cj0(A) = Cj0\cdot Area, \quad \text{and} \quad r_s(A) = rs/Area. \quad (16.30)$$

The test circuit shown in Fig. 16.23 illustrates how device capacitance and resistance can be determined as a function of diode bias voltage. Firstly, the diode S parameters are determined at a given bias voltage, secondly these are converted to Y parameters and the diode capacitance (Cap) and resistance (RD) extracted from $Y[1,1]$, and finally the variation of Cap and RD with diode voltage V_d plotted using the Qucs plotting function PlotVs. Notice that the value of Cap at $V_d = 0V$ agrees with the value of $Cj0$.

To complete the demonstration EDD diode model all that remains to do is to add temperature dependence to the current and capacitance equations. Circuit simulators normally use two temperatures to determine device temperature dependence; the first called Tnom represents the temperature that the device parameters were measured, and the second called Temp represents the current device temperature. A high percentage of the diode parameters are temperature dependent. However, to simplify the demonstration diode model only the temperature dependence of parameters I_s, Vj and $Cj0$ will be included

[40] In complex current and charge expressions precalculating subexpressions in equation blocks ensures that they are only calculated once at the beginning of a simulation, ensuring minimum run times for an EDD model.

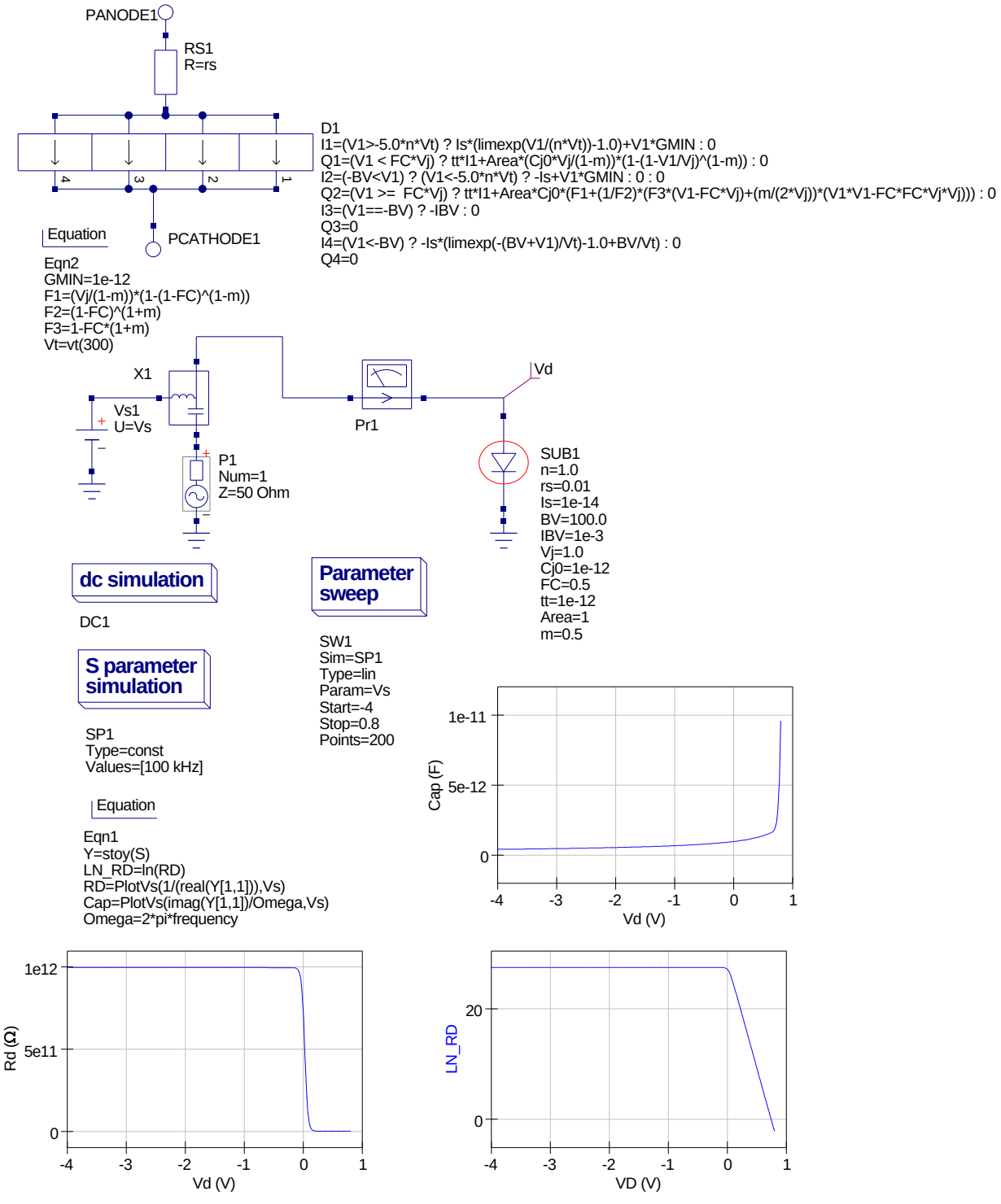


Figure 16.23: Compact diode model capacitance and resistance simulation

in the model. Adding extra temperature dependence to the diode model is left to readers as an exercise[41]. One of the great advantages of the EDD style of modelling is that it is interactive allowing easy experimentation with models to any given level. The following equations list the temperature dependence of I_s, Vj and $Cj0$.

Let T1 = Tnom and T2 = Temp, then

$$I_s(T2) = I_s(T1) \left\{ \frac{T2}{T1} \right\}^{\frac{XTI}{n}} \exp \left[\frac{-q \cdot Eg(300)}{kB \cdot T2} \left(1 - \frac{T2}{T1} \right) \right] \tag{16.31}$$

$$Vj(T2) = \frac{T2}{T1} \cdot Vj(T1) - \frac{2 \cdot kB \cdot T2}{q} \ln \left(\frac{T2}{T1} \right)^{1.5} - \left[\frac{T2}{T1} \cdot Eg(T1) - Eg(T2) \right] \tag{16.32}$$

$$Cj0(T2) = Cj0(T1) \left[1 + m \left\{ 400 \cdot 10^{-6} \left(T2 - T1 \right) - \frac{Vj(T2) - Vj(T1)}{Vj(T1)} \right\} \right] \tag{16.33}$$

In these equations:

- XTI = Saturation current temperature exponent.
- $Eg(T) = EG(0) - \dfrac{7.02e-4 \cdot T^2}{1108 + T}$, the energy gap.

Figure 16.24 shows the extended EDD for the experimental diode model. Again the limexp() function is used in preference to the standard exp() function in the temperature calculations listed in equations block Eqn2. The test circuit in Fig. 16.24 sweeps the device temperature from 20 to 80 degrees Centigrade. The graph inlay illustrates the experimental diode current I_d plotted as a function of temperature. The temperature of the built-in Qucs diode is held constant, at room temperature, and it's current Id_Q plotted as an overlay. The two curves cross at room temperature, indicating identical currents at this temperature.

[41] For example, parameters m and BV are both temperature dependent.

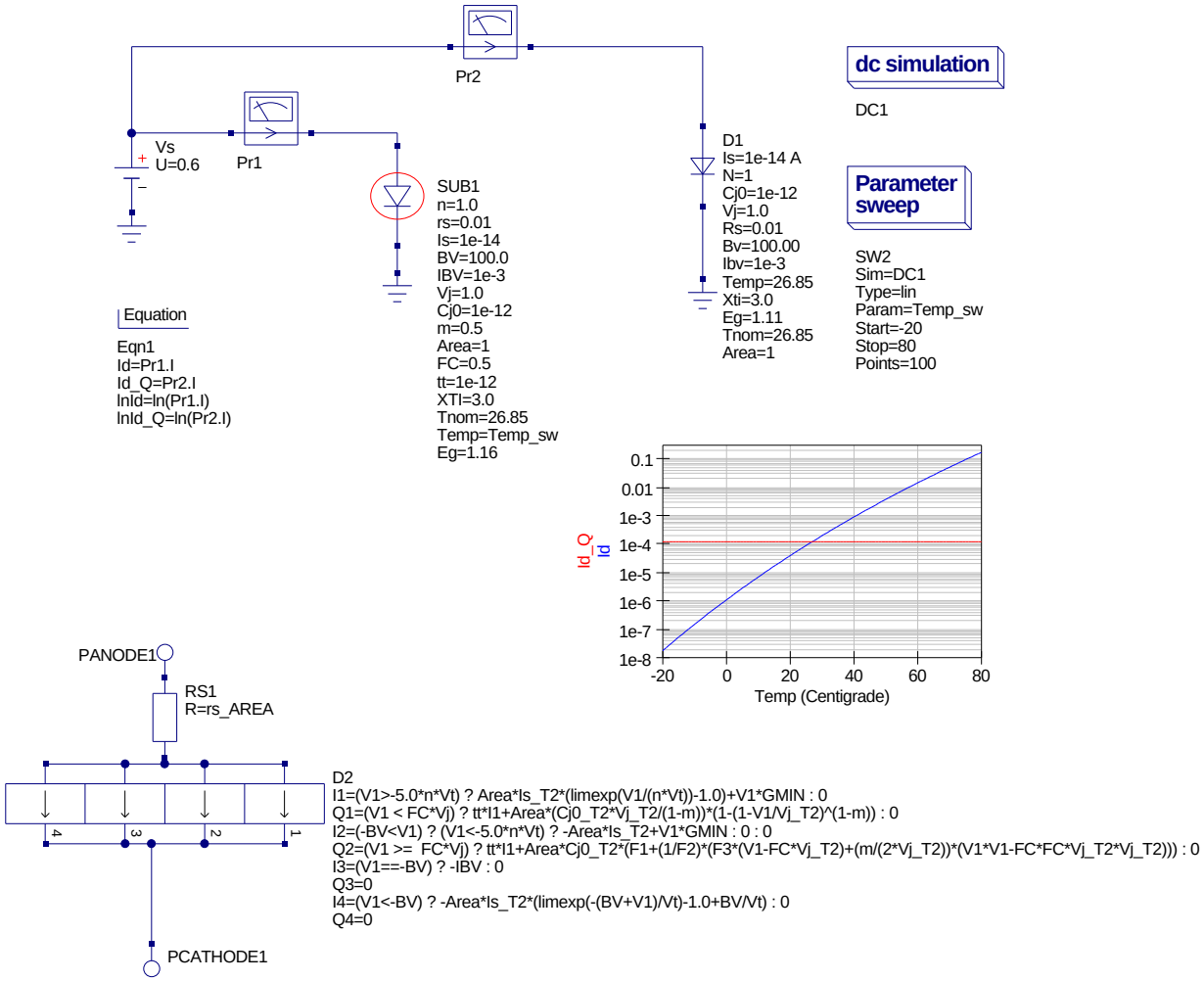


Figure 16.24: Compact diode model with temperature dependence

16.12 Constructing EDD compact device models and circuit macromodels

Component equations, subcircuits with parameters and EDD models are major developments for the Qucs circuit simulator. They provide advanced modelling capabilities with enough power and flexibility to allow a much greater range of models to be developed than the ones currently provided with each Qucs release. In the future it is proposed to add new models to the Qucs Web site. The Qucs team is very keen to encourage all Qucs users to support the modelling effort. If you have constructed a new model and would like to share it with other Qucs users please post your model on the qucs-devel or qucs-help mailing lists. Both the model schematic file and a brief outline of its operation and specification are requested. An example model specification for the Curtice MESFET device can be found on the Qucs Web site. Please use the same format when writing model descriptions.

16.13 End Note

This tutorial note introduces a large number of new modelling concepts and shows how equations, subcircuits with parameters and the new equation defined device perform a central role in constructing Qucs models. The EDD approach to modelling makes possible, for the first time, the construction of equation defined compact device models and circuit macromodels using the Qucs schematic capture facilities as an interactive modelling medium. This is a major step forward for Qucs. Once again these notes are very much a record of work in progress: much still remains to be done in the future to improve the modelling capabilities provided by Qucs. A major short term task will be the development of additional models covering as wide a range of applications as possible. If Qucs is to fulfill it's mission to become a truly universal circuit simulator then it must be supported by models. Some readers will have noticed that these notes include very little information about the ADMS-Verlog-A and hand coded C++ model development routes. This was a deliberate decision on my part. Sometime in the future I intend to return to these subjects and update the tutorial. A very special thank you must go to Stefan Jahn for all his hard work, skill, and dedication during the period he has worked on programming the amazing modelling capabilities now embedded in Qucs.

16.14 Appendix A: Qucs constants, operators and functions

This appendix lists the constants, operators and a number of functions that are available for constructing Qucs equations. Items in [...] indicate the equivalent object in the Verilog-A language. The functions listed are common to Qucs and Verilog-A. A number of other functions have been implemented in Qucs. The full list can be found in the Qucs help system; "Short Description of mathematical Functions" or in the Qucs "Measurement Expression Reference Manual" by Gunther Kraut and Stefan Jahn, `http://qucs.sourceforge.net/docs.html`.

- Constants
  1. pi = 3.141593...
  2. e = 2.718282...
  3. kB = 1.380651e-23 J/K
  4. -q = -1.602177e-19 C
- Operators
  1. +x unary plus
  2. -x unary minus
  3. x+y addition
  4. x-y subtraction
  5. x*y multiplication
  6. x/y division
  7. x%y modulo (remainder)
  8. x^y power [pow(x,y)]
  9. ?: ternary (condition) ? (expression if true) : (expression if false)
  10. || logical or
  11. && logical and
  12. == equal
  13. < less than
  14. <= less than or equal to
  15. > greater than
  16. >= greater than or equal to
  17. != not equal to
  18. ( ) brackets

- Functions
  1. ln(x) natural logarithm
  2. log10(x) decimal logarithm [log(x)]
  3. exp(x) exponential function base e
  4. sqrt(x) square root
  5. min(x,y) minimum
  6. max(x,y) maximum
  7. abs(x) absolute value
  8. sin(x) sine
  9. cos(x) cosine
  10. tan(x) tangent
  11. arcsin(x) inverse sine [asin(x)]
  12. arccos(x) inverse cosine [acos(x)]
  13. arctan(x[,y]) inverse tangent [atan2(x,y)]
  14. sinh(x) hyperbolic sine
  15. cosh(x) hyperbolic cosine
  16. tanh(x) hyperbolic tangent
  17. arsinh(x) inverse hyperbolic sine [asinh(x)]
  18. arcosh(x) inverse hyperbolic cosine [acosh(x)]
  19. artanh(x0 inverse hyperbolic tangent [atanh(x)]
  20. limexp(x) argument limited exponential function
  21. hypot(x,y) Euclidean distance function

16.15 Appendix B: Constructing subcircuits with parameters

In this appendix a series of screen dumps illustrate the sequence needed to construct a subcircuit with parameters. A simple series resonance circuit has been chosen for the demonstration.

16.15.1 Enter the series resonance circuit and add input and output pins

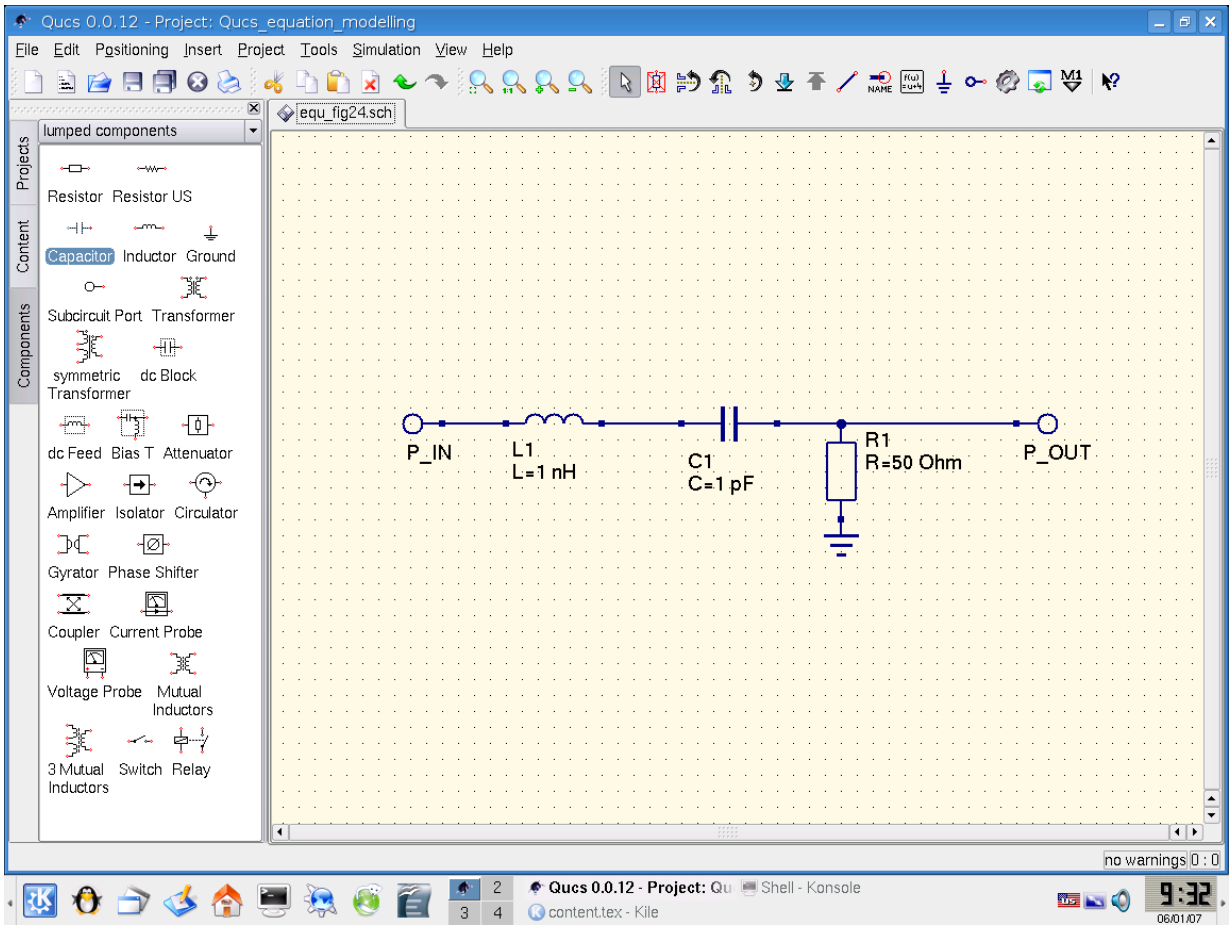


Figure 16.25: Stage 1: screen dump showing LCR circuit

16.15.2 Change the component names to Ls, Cs and Rs

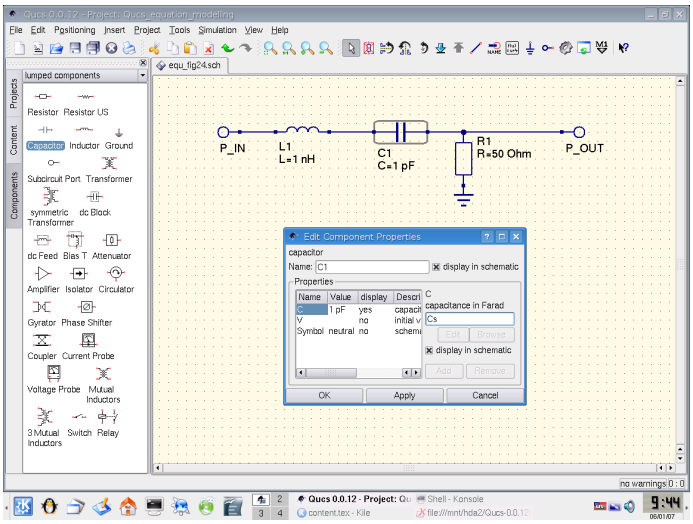


Figure 16.26: Stage 2: screen dump showing LRC circuit

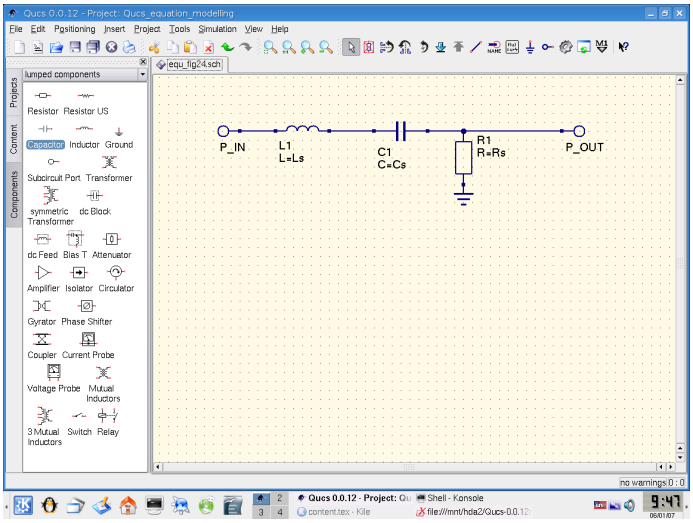


Figure 16.27: Stage 2: screen dump after name changes

16.15.3 Construct symbol for new subcircuit

Right click on the Qucs drawing area and select Edit Circuit symbol or press key F9. Edit the drawing symbol to give the design shown in Fig. 16.28.

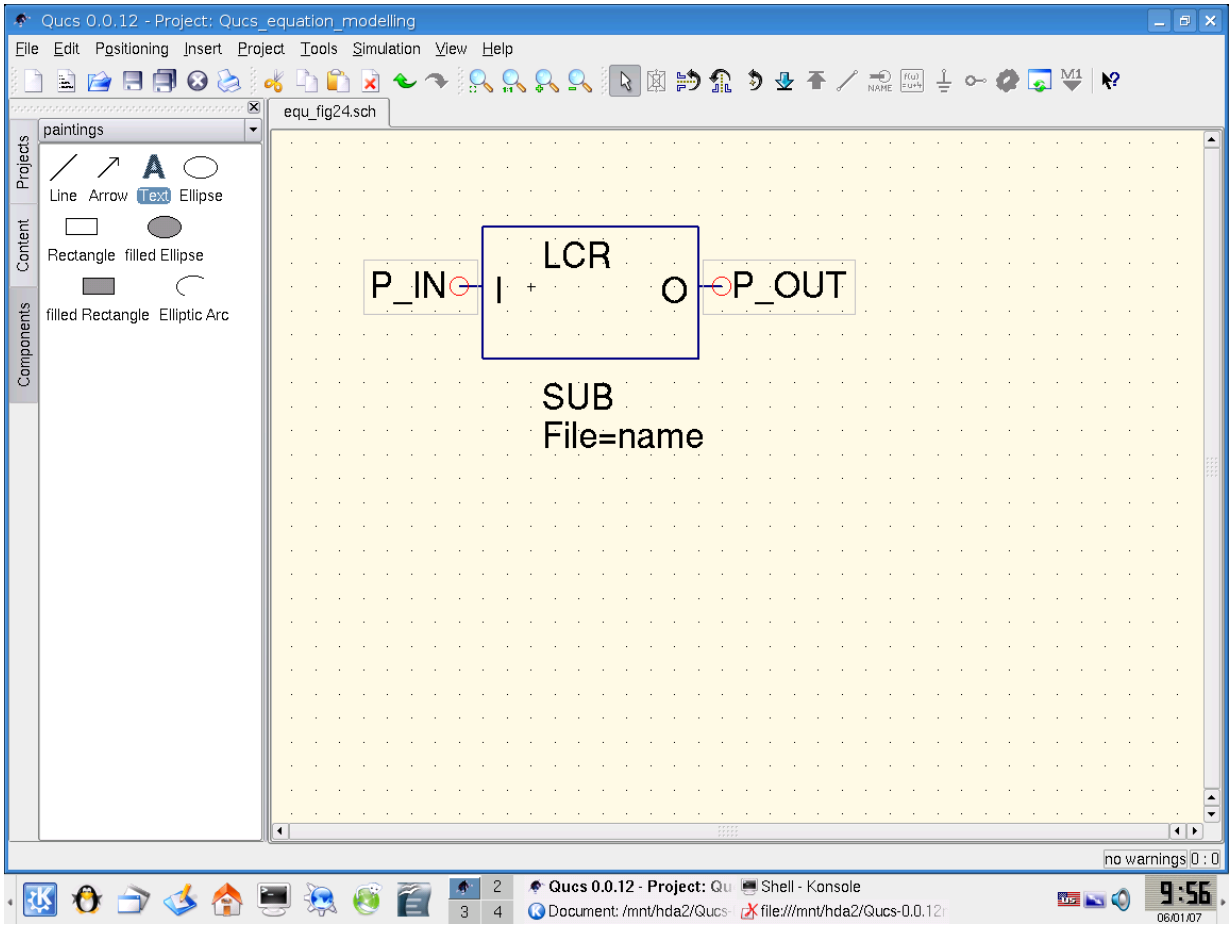


Figure 16.28: Stage 3: the subcircuit symbol

16.15.4 Add the names of the subcircuit parameters to the LCR symbol

Right click on the SUB / File=name caption and enter names of subcircuit parameters with their default values.

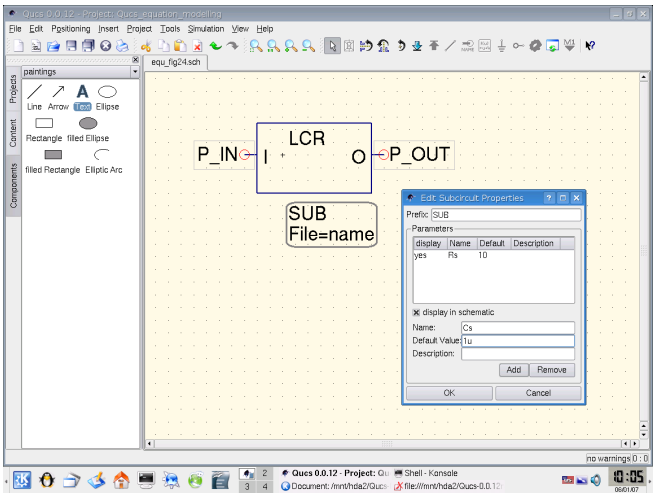


Figure 16.29: Stage 4: entering subcircuit parameter names and default values

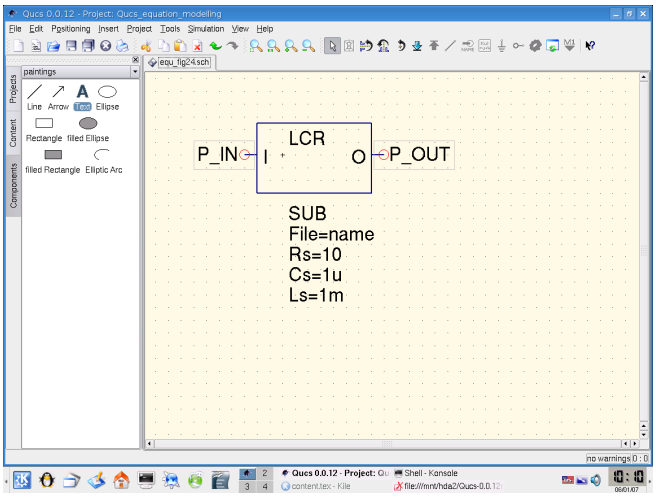


Figure 16.30: Stage 4: resulting subcircuit and parameter list with default values

16.15.5 Test the LCR subcircuit

Figure 16.31 gives a simple AC transfer function test circuit and resulting waveforms. Parameter R_SW is swept over the range 1Ω to 10Ω and the AC transfer function recorded and plotted.

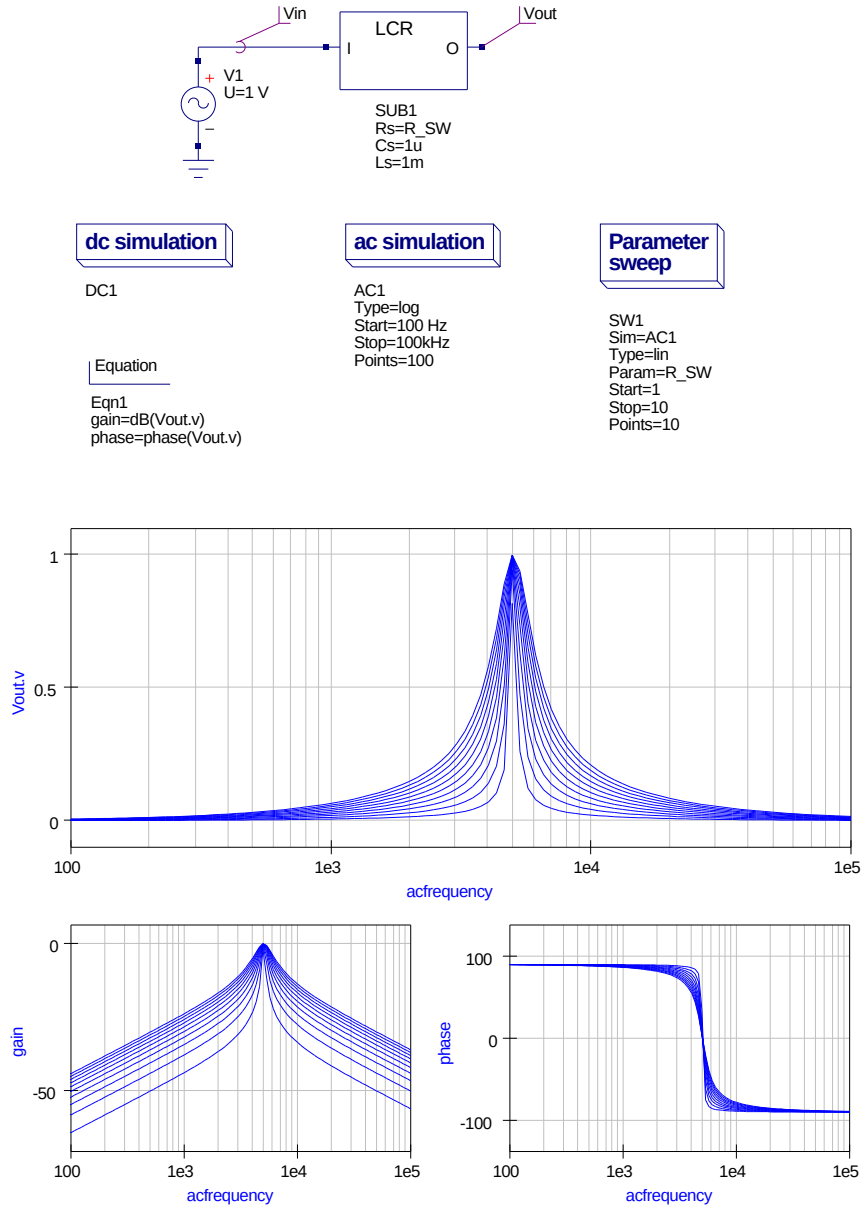


Figure 16.31: Stage 5: Subcircuit test circuit and output waveforms