

# ModelSim Tutorial

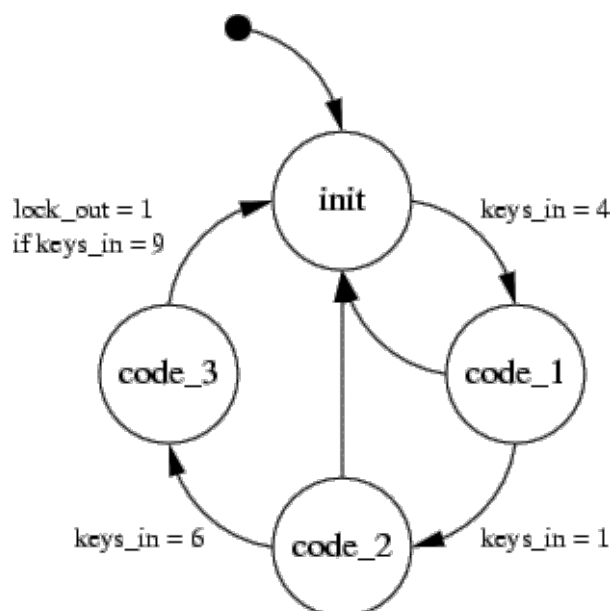
## Getting Started

### Introduction

ModelSim is an easy-to-use yet versatile VHDL/(System)Verilog/SystemC simulator by Mentor Graphics. It supports behavioral, register transfer level, and gate-level modeling. ModelSim supports all platforms used here at the Department of Pervasive Computing (i.e. Linux, Solaris and Windows) and many others too. On Linux and Solaris platforms ModelSim can be found preinstalled (see [Linux/Mustatikli](#)) on Department's computers. Windows users, however, must install it by themselves.

This tutorial is intended for users with no previous experience with ModelSim simulator. It introduces you with the basic flow how to set up ModelSim simulator, compile your designs and the simulation basics with ModelSim SE. The example used in this tutorial is a small design written in VHDL and only the most basic commands will be covered in this tutorial. This tutorial was made by using version 6.1b of ModelSim SE on Linux.

The example used in this tutorial is a simple design describing an electronic lock that can be unlocked by entering a 4-digit PIN (4169) code from a key pad. When the lock detects the correct input sequence, it will set its output high for one clock cycle as a sign to unlock the door. The figure below shows the state machine of the design. The design also includes one dummy variable (`count_v`) which has no practical meaning but is used to demonstrate debug methods in ModelSim.



Lock's state machine diagram.

When reset, the state machine starts from the state 'init' in which it'll wait until it reads number 4 from `keys_in` bus and moves to state 'code\_1'. Then, if the next two numbers on `keys_in` bus are 1 and 6, the state machine proceeds through the state

'code\_2' to state 'code\_3'; otherwise it returns back to state 'init'. From the state 'code\_3' the state machine always returns back to state 'init'. If the code on `keys_in` bus is 9, the state machine also sets its output high for one clock cycle.

# 1. Initialization (Linux/Solaris only)

In this section we will create a working directory for our design and then a design library in which the design is compiled. But before ModelSim can be used your environment must be set up. This includes setting a few variables, files and directories and can be done by sourcing the scripts given below. (Note that autocompletion of file names does not necessarily work in directory /share)

Windows users may skip this step and jump directly into section [Creating Libraries](#); there are no scripts to be run before using ModelSim on Windows.

## a) Linux

Set up the environment by writing the following to the shell window in Linux. Note that file path completion does not necessarily work under /share (reason is a bit unclear)

```
$ source /share/tktprog/mentor/modeltech-6.3d/modeltech.sh
```

## b) Solaris

First, check this link ([Mustatikli](#)) to find out the latest version of ModelSim currently available on Solaris; then source it using the command given on the selected page. Select the latest version available unless you have a reason to use a specific version.

```
$ source /opt/mentor/modeltech-6.3a/modeltech.sh
```

After sourcing the given script you should see a message similar to the following indicating that the source script was read and your environment was set up correctly:

```
#####  
ModelSim SE version 6.3a  
#####
```

## 1.1 Creating Libraries

First, to keep everything (project files and directories etc.) nicely organized and separated from other projects and files, create a directory for your project if you don't already have one and change into that directory:

```
$ mkdir ex1_tutorial  
$ cd ex1_tutorial
```

Then, in your newly created project directory, create a design library and map it. Design library is a library in which ModelSim stores your compiled design units. Mapping is required so that ModelSim can locate the design library. The default design library is referred as `work` in VHDL files. Symbolic name `work` is mapped to directory `./work` by default. Here we map it to directory `my_lib`.

```
$ vlib my_lib          # Create your own design library ...
$ vmap work $PWD/my_lib # ... and map it as 'work'
Copying /share/tktprog/mentor/modeltech-6.1b/linux/./modelsim.ini to modelsim.ini
Modifying modelsim.ini
```

As you can see, ModelSim creates a directory `my_lib` into your current directory and a file `_info` into it. You should never edit this file by yourself. It is used by ModelSim to keep track about all design units and their state in your design. You may consider it like a table of contents of the design library for the simulator.

In mapping, ModelSim copies a file called `modelsim.ini` in your current directory (in this case `ex1_tutorial`) if it is not there already and modifies its library section. When ModelSim is invoked, it will read this file and use its mappings to locate design libraries. You may want to check the contents of `modelsim.ini` for the current library mappings by opening it in text editor or by invoking command `vmap` without any arguments:

```
$ vmap
Reading modelsim.ini
"work" maps to directory my_lib.
Reading /share/tktprog/mentor/modeltech-6.1b/linux/./modelsim.ini
"std" maps to directory /share/tktprog/mentor/modeltech-6.1b/linux/./std.
"ieee" maps to directory /share/tktprog/mentor/modeltech-6.1b/linux/./ieee.
...
```

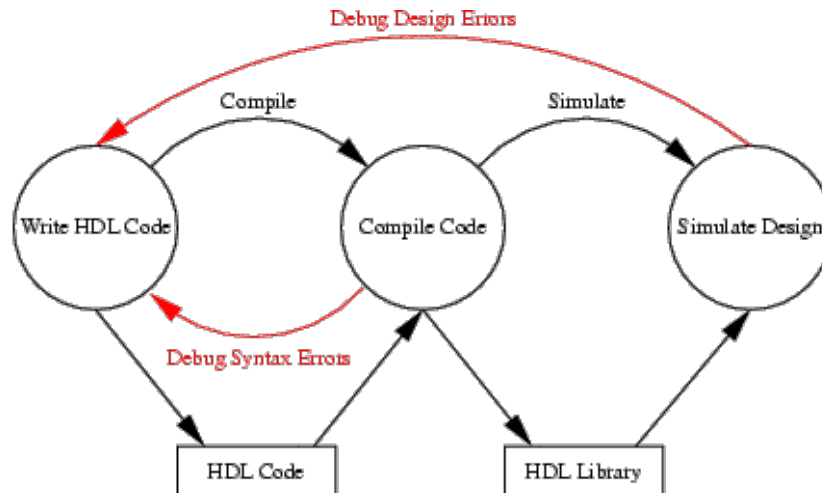
You may have any number of design libraries wherever in your file system you want as long as they are correctly mapped in your `modelsim.ini` and ModelSim can find them. E.g. you can place them into `/tmp/$USER/` to save quota; just do not store your source codes in `tmp`.

Similarly, you can compile source files from multiple locations into the same design library.

## 2. Compiling the Design

Now you have your environment set up and a project directory `ex1_tutorial` as well as a design library `my_lib` created. Since the simulator can't read VHDL (or Verilog) source code itself, the next step is to compile your designs into your design library. Note that VHDL is strict about the order in which VHDL files are compiled. That is, a file or files that are referenced by other files (such as packages) must be compiled before files referencing to them (e.g. files using those packages).

During the compilation all syntactical bugs will be pointed out. After a succesful compilation, actual simulation/debugging may begin. You will see pretty soon that the design cycle will be something like `edit - compile - simulate - edit ...` as shown in figure below.



A common design cycle. After the HDL file has been written it will be compiled and syntactical errors are fixed. Then the compiled code is simulated and functional errors debugged.

First, create a separate directory for source files in your project directory (ex1\_tutorial/) and

```
$ mkdir vhd
```

Second, download the following VHDL files there (right mouse button -> save as OR use \$ wget with link(s) below as argument)

- [lock\\_pkg.vhd](#)
- [lock.vhd](#)

Now it's time to compile the sources. Use command line option '-check\_synthesis' to identify structures that may not be synthesizable. Note that the -check\_synthesis option checks compliance at least for some most common synthesis rules (but not necessarily all).

```
$ vcom -check_synthesis vhd/lock_pkg.vhd
$ vcom -check_synthesis vhd/lock.vhd
OR
$ vcom -check_synthesis vhd/lock_pkg.vhd vhd/lock.vhd
```

You can compile source files one by one or all in one command as shown above. In any case, they must be compiled in correct order: file lock.vhd must be compiled last since it uses the package file.

If you are working in a project with lots of files you will soon notice that managing the compilation of all those files will get too cumbersome (that is, editing one package file might require a recompilation of other files which would again require recompilation of other files etc.). In UNIX, such process can be automated using make. You'll need a file called Makefile that describes the relationships between the files and the commands how to update them. Once you have such a file, you can recompile only the required design units (i.e. those that you have modified and those which are dependent on recompiled design units) just by calling make.

Luckily, you don't have to know anything about writing makefiles; ModelSim comes with a tool for that: vmake. vmake creates automatically a makefile for a design library specified on command line. Note that vmake uses the information found from the

design library to create a makefile for that library (i.e. you'll have to compile all your files once manually as shown above before you can call `vmake` to create a makefile).

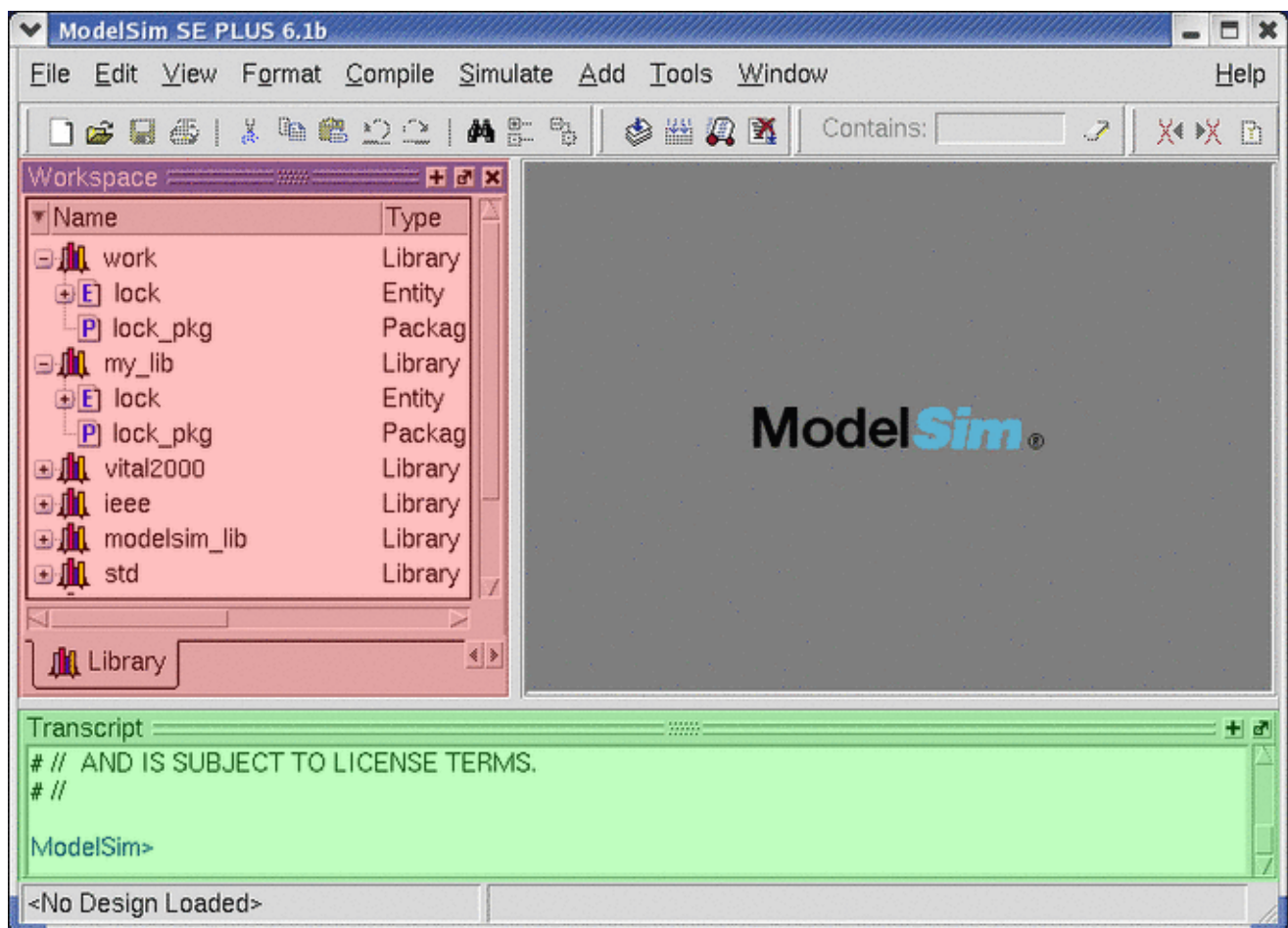
```
$ vmake my_lib > Makefile
```

Now you can keep your design library up-to-date simply just by calling `make`. If you need to add new files in your design, you can do it by compiling them into your design library and then regenerating your makefile. Similarly, files can be removed with command `vdel` (removes files from design library) and then regenerating your makefile again. **Note!** Vmake command does not seem to work in Windows for unknown reason.

## 3. Simulation

Now it's time to simulate your design. ModelSim simulator is invoked with command `vsim`. The default window will look something like the picture below. There might also be a welcome window when launched for the first time.

```
$ vsim &
```



Default Simulator window. Red shows workspace pane and green Transcript pane.

The picture above shows library `work` and library `my_lib` opened in Workspace pane. Since library `my_lib` was mapped to default library (`work`) while `my_lib` was created, these two libraries are actually just one and the same library.

Transcript pane shows the messages between the simulator (e.g. errors encountered by the simulator or messages printed by the design/testbench) and the designer (commands entered to `ModelSim>` prompt). Messages in Transcript pane are recorded in a file called `transcript`.

Other panes include `Wave` window to view waveforms, `Dataflow` window, `source` window etc. They can be found from Menubar (`view`) if needed.

## 3.1 Load the Design

First thing to do is to load the design into the simulator. After ModelSim is invoked there are usually two or three ways to execute a command: commands can always be written to ModelSim prompt in Transcript pane. Some of the commands can also be found from toolbar buttons or from menubar dialogs.

For example, to load the design 'lock' from library `my_lib`, you can:

- open the library `my_lib` in Workspace pane and double click the entity named `lock`
- select `Simulate - Start Simulation...` from Menubar and then similarly as above, select the library and entity you want to simulate
- write `vsim my_lib.lock` in ModelSim prompt (or start simulator with that command from command line)

In this tutorial we will mostly write commands in Transcript pane as they will be recorded into the `transcript` file. Once that we have finished the simulation, we can use the `transcript` file to create a macro file that can be reused in subsequent simulations.

```
ModelSim> vsim my_lib.lock
```

You can also provide command line arguments but we won't need them in this tutorial. However, the newest Modelsim versions are sometimes optimizing too greedily and you won't necessarily see all the signals. In those cases, just disable optimizations: `vsim -novopt my_lib.lock`

## 3.2 Prepare Simulation

After loading the design into the simulator but before a simulation can be started, you need to set up your simulation/debug environment for the design. That is, you need to open the debug windows you consider important and select signals and variables etc. you want to trace on those debug windows.

A good choice in the beginning for debug windows include `Wave`, `Source`, `Locals` (variables) and `Objects` (signals) pane. These can be opened from the pull-down menu `view` or from command line

- `Wave` - See waveforms
- `Objects` - See signals and their values
- `Source` - Trace your code line-by-line
- `Locals` - See variables and their values



```

VSIM> view objects
VSIM> view locals
VSIM> view source
VSIM> view wave -undock # Detach wave as a separate window (undock)

```

Next, you need to tell the simulator which signals you want to trace in wave window. If you have lots of signals it might be a good idea to select only those signals that you are interested in, but in this case we will be tracing them all:

```
VSIM> add wave * # Show all signals that are in Objects pane in Wave window
```

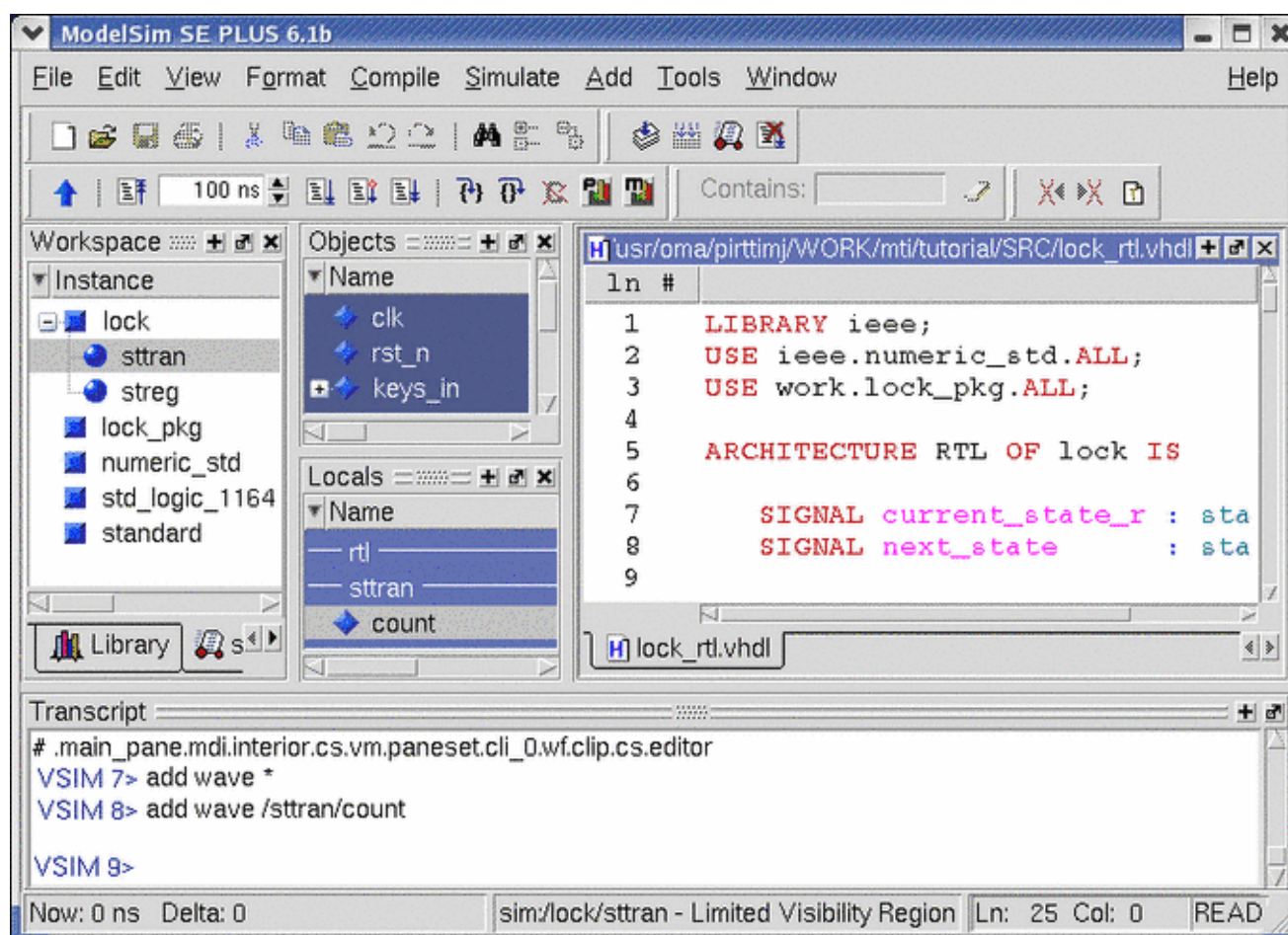
You can also add signals into Wave window:

- Select signals in Objects pane (Shift+Left Mouse) and drag'n'drop the selection to Wave pane.
- Right click in Objects pane and select Add to Wave.
- Add to Wave dialog can also be found from menubar (Add).

There is also one variable (count\_v) which we will be tracing on Wave window:

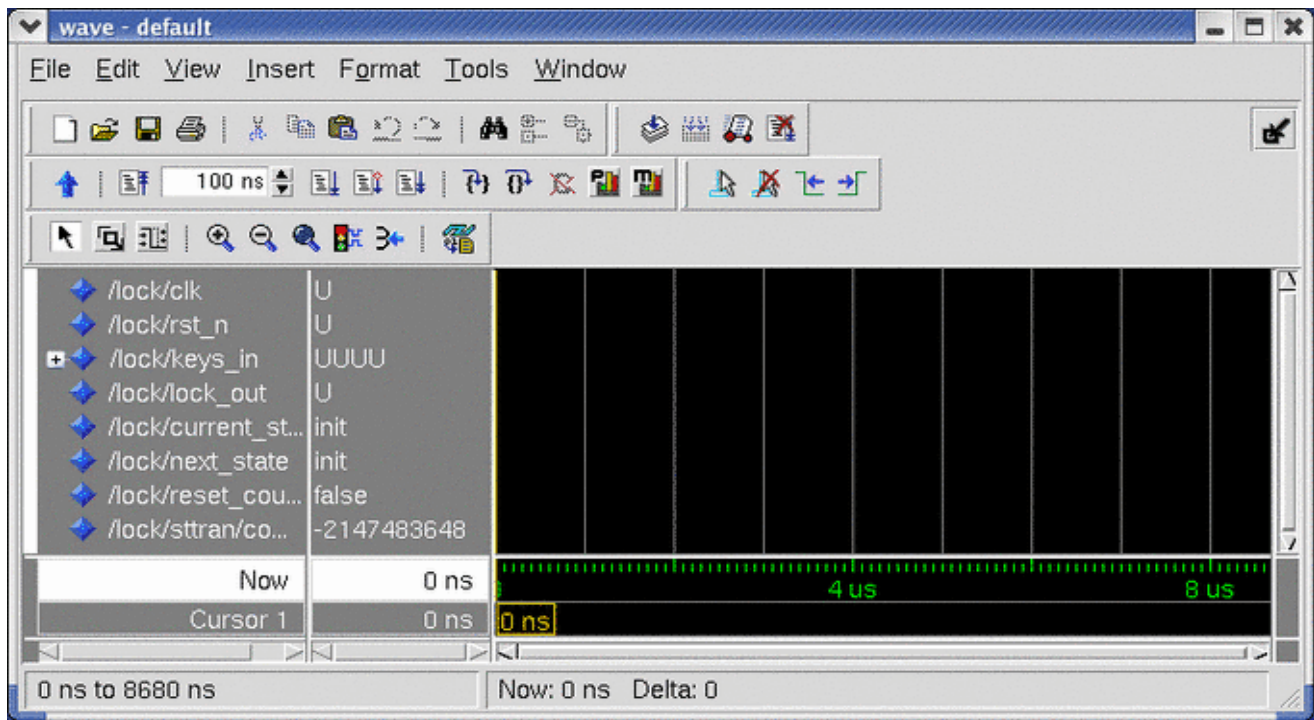
```
VSIM> add wave /sttran/count_v
```

Now everything should look something like below:



ModelSim Main window showing Workspace, Objects (signals), Locals (variables) and Source panes. There are two processes in this design (sttran and streg), shown in Workspace pane. Locals pane shows variables (count\_v) in process currently selected in Workspace pane (sttran).





Wave Window. Values shown for signals and variables are still uninitialized.

It is very beneficial to save current signal configuration of the wave window by selecting **File->Save...**

There is still one more thing to do before we can start the simulation: we need to generate the input stimuli for the design. There are three inputs in our design: asynchronous, active low reset (`rst_n`), system clock (`clk`) and a bus from the keypad (`keys_in`). We will generate the following input stimuli for simulation start up: system clock (50 MHz and 50/50 ratio), reset (active for 45 ns) and input bus set to 0 (0000).

```
VSIM> force -deposit /rst_n 0 0, 1 {45 ns}
VSIM> force -deposit /clk 1 0, 0 {10 ns} -repeat 20
VSIM> force -deposit /keys_in 0000 0
```

The last line is easiest to understand of these three lines, it just sets the value of `keys_in` bus to 0000 at 0 ns. The first line is bit more complex, but it generates a reset signal that is 0 at time 0 ns and goes high at 45 ns. The second line generates the system clock. First, the clock is set high at 0 ns (1 0) and 10 ns later it goes low (0 {10 ns}). This is the first half of the clock cycle. For the second half of the clock cycle, clock stays low until it is changed again. Since the command is repeated/starts over every 20 ns (-repeat 20), this command will eventually generate a clock signal that is high for the first half of the clock cycle and low for the second half and that has a period of 20 ns.

Note: the default time unit (Resolution) is defined in `modelsim.ini` and is ns. We could therefore omit time units altogether and write two first commands as follows (-repeat can also be written as -r):

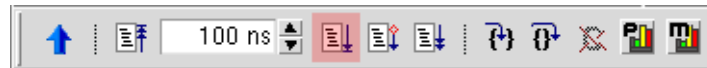
```
VSIM> force -deposit /rst_n 0 0, 1 45
VSIM> force -deposit /clk 1 0, 0 10 -r 20
```

Now we are ready to do the first simulation of our design.

Note: In the real world, the clock signal is more easily assigned in the test bench and not by hand. However, in some cases these commands come in useful.

## 3.3 Run Simulation

All inputs have been set with initial values and everything is ready for a simulation. To run the simulation press the `Run` button. Run button can be found from both ModelSim Main and Wave window as well as from menubar (`simulate -> Run -> Run 100ns`).



Run button in ModelSim Main/Wave Window Toolbar.

As always, you can also write the same command in ModelSim prompt:

```
VSIM> run <time>
```

Now run the simulation for a few hundred nanoseconds. You should see how each signal takes at start up their initial values set in previous step and how reset deactivates after a few clock cycles. Next, we will try enter the correct PIN code (4169) to see how the design reacts with that.

Some designs produce huge number of ugly-looking warnings at 0 ns, such as Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es). They are due to number conversion functions whose inputs are not real numbers, e.g. undefined 'U', and they make it harder to spot the more important warnings. You can get rid of them, by starting simulation like this:

```
vsim mylib.lock
set StdArithNoWarnings 1
run 0 ns
set StdArithNoWarnings 0
run 100
```

Note that they warnings are enabled again. You can also edit `modelsim.ini` to disable these warnings permanently, but usually we wish to see them when if they occur after 0 ns.

### 3.3.1 Input generation

To simulate the key presses we have to change the `keys_in` inputs on each clock cycle:

```
VSIM> force -deposit /keys_in 2#0100 # set keys_in bus to 4 ...
VSIM> run 20 # ... and run for one clock cycle
VSIM> force -deposit /keys_in 2#0001 # 1
VSIM> run 20
VSIM> force -deposit /keys_in 2#0110 # 6
VSIM> run 20
VSIM> force -deposit /keys_in 2#1001 # 9
VSIM> run 20 # lock_out should be active now
VSIM> force -deposit /keys_in 2#0000 # 0 (for "clearing" the bus)
VSIM> run 100
```

Entering the inputs like shown above is of course a tedious job. First, you need to write 'force -deposit ...' for each cycle and then run the simulation forward one clock cycle. Second, you need to enter the inputs in binary format. It is doable with few bits, but not very practical with 32 bits or more.

There is, of course, an alternative and faster way to write the above commands:

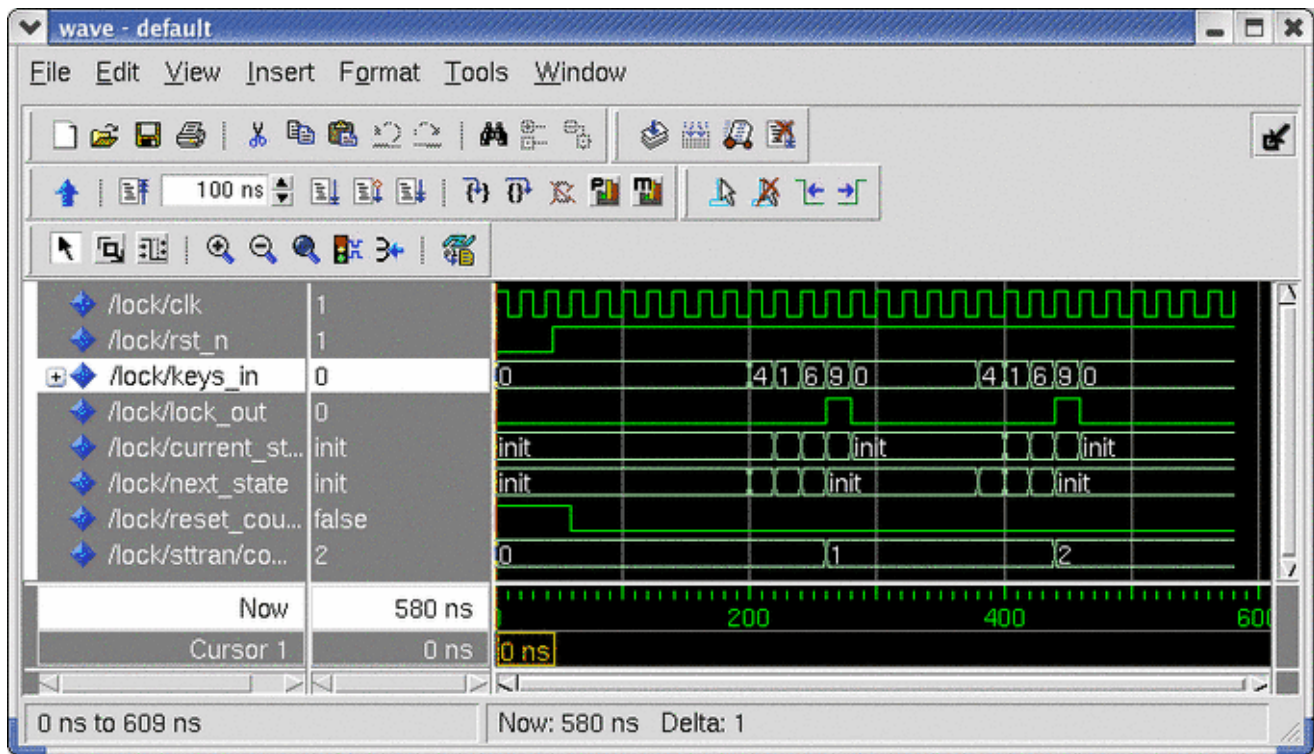
```
VSIM> force -deposit /keys_in 10#4, 10#1 20, 10#6 40, 10#9 60, 10#0 80
VSIM> run 200
```

The above line simply says: change the `keys_in` bus to decimal value 4 (10#4) starting from time instant now. Then, 20 ns later, change the input to 1 (10#1 20), 40 ns later to 6 (10#6 40), 60 ns later to 9 (10#9 60) and finally 80 ns later back to 0 (10#0 80). In addition that everything is done in one line, it also shows how inputs can be entered in decimal format. Other possible radices that can be used are binary (2#), octal (8#) and hexadecimal (16#).

For clarity it is sometimes also good to change the radix for some signals shown in waveform window. For example, if you have wide buses in your design, it will be easier to check the values on those buses if data is shown e.g. in hexadecimal format rather than in binary format. Changing the radix for a signal can be done by right clicking the signal in Wave window and selecting `Radix` and then the correct radix from the dialog box. In this case correct radix is unsigned.

```
VSIM> property wave -radix unsigned /lock/keys_in
```

Now the Waveform window should look something like below:



Results for the first simulation run. Radix for `keys_in` bus has been changed to unsigned.

The design seems to be working as it should, but so far we have tested just one input sequence. Now we will try a few other input sequences to see how the design deals with them.

First we will try a random input sequence that also contains the correct sequence embedded inside the random sequence (25416918):

```
VSIM> force -deposit /keys_in 10#2, 10#5 20, 10#4 40, 10#1 60, 10#6 80, 10#9
100, 10#1 120, 10#8 140, 10#0 160
VSIM> run 300
```

As you can see, the code is accepted and door unlocked if there is a long enough random input sequence containing the correct code. If this is an undesired feature for a lock, you'll have to redesign the system.

Next we will try a partially correct code (4119) as an input sequence:

```
VSIM> force -deposit /keys_in 10#4, 10#1 20, 10#9 60, 10#0 80
VSIM> run 200
```

Again the input sequence was accepted even though it's clearly a wrong code. To find out what went wrong, we'll have to debug our design.

### 3.3.2 Creating a macro (.do) file

But before we start debugging the code, we will create a macro file from what we have done so far. Macro (do) files are files that contain ModelSim and sometimes Tcl commands to control the simulator and the simulation. Macro files are useful files that can help you to reduce repetitive work like setting up the simulator (open debug windows) or simulation (initialize signals) and so on. You can even use them run the whole simulation with one command. Once you have finished the

simulation, create a macro file from the commands you have used and then you can run the same simulation just by calling the macro file. It is very useful, since you always modify your design and you must simulate it again with the same data as before.

Like said earlier, all commands that you have written in ModelSim prompt (and some of those which have been selected through menu/toolbar buttons) have been recorded in file `transcript`. We will, however, modify the file a bit before we will use it. You might have made, for example, typos when writing some commands and there are also some comments written by ModelSim which will be removed.

So, use your favorite text editor and edit a file named `transcript`:

- remove comments from the file
- add line `'delete wave *'` before line `'add wave *'`
- fix typos (if any)
- remove the last `run` command

Rename your new macro file, for example, as `lock.do` (or it will be overwritten by new `transcript` file). Your new macro file should look something like this example of [lock.do](#).

Line `delete wave *` above is required since otherwise signals in your design will be added to Waveform window again each time simulation is restarted. You could also divide the macro file in two parts -- `startup.do` and `lock.do` -- and put in the first file all the stuff that can be done at simulator startup (opening windows and adding signals/variables in Waveform window) and in the second all the rest. Then uncomment the line `; Startup = startup.do` in file `modelsim.ini` and quit and restart simulator. Now whenever simulator is invoked in this directory it will execute the `startup.do` file and execute the commands in there.

Now you can run the whole simulation we have done so far just by calling your macro file:

```
VSIM> restart -f
VSIM> do lock.do
```

And since we removed the last `run` command after the defective input sequence, the simulation is now set up for debugging the error.

## 3.4 Debugging the Design

Though the error in this design can easily be found just by reading the code, we will look at what kind of methods ModelSim provides for debugging the design. These methods include: examining and setting signals and variables values, setting breakpoints and executing the code line by line, setting checkpoints to restore simulation state and so on.

Note that before an error can be debugged, it must be detected and repeated. In general, one should automate both stimulus generations and response checking.

### 3.4.1 Examining Signals in wave window


First we will look at how the values of signals and variables can be examined during the simulation.

Signals can be drag-and-dropped and copy/cut-pasted to get them organized properly. For example, clock signal is usually copied few times. Note that ordering and proper radix have a major impact on readability and hence debuggability. It is recommended that signals activity "flows from top left to bottom right".

You may group signals according to their purpose by adding *divider* texts to the signal list.

Select a signal then click right mouse button or select from menus `Insert -> Divider`. Dividers help working with large number of signals.

Note that there is 4 different zooming options: "in", "out", "full" (shows the whole run in one view), and "in on active cursor". The last one keeps the active cursor in the middle of window which is handy.

When you have selected a signal from wave window, you search for (falling and rising) edges with these buttons .

Selected signal can be forced to new value for debug purposes with right-mouse button-> Force. You may also search for certain value. This is especially suited for multibit signals (=buses).

Select from menus `Edit -> Signal Search -> Search for Signal Value...`

First letter denotes the radix: B=binary, X=hexadecimal, and the value is given inside quotation marks, for example `x"3"`

### 3.4.2 Examining and Setting Signals and Variables with commands

You can always put the signals and variables on Waveform window, but for any larger design there will probably be more than enough data to be shown at one time. The `examine` command can be used to examine the values of both signals and variables in your code:

```
VSIM> examine curr_state_r
# init
VSIM> examine /lock/sttran/count_v
# 3
```



Note that signal `curr_state_r` is specified in the current context (`sim:/lock`) but variable `count_v` is not (it's defined in context `sim:/lock/sttran`; i.e. inside the process `sttran` of architecture of the `lock`). Therefore `curr_state_r` does not need path, but `count_v` needs to be specified with path to it.

You can also examine past values of objects by specifying time instant as well as set the radix (since the current context is `sim:/lock`, we can also specify path to `count_v` relative to the current context):

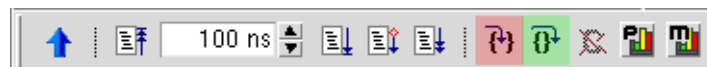
```
VSIM> examine -time 675 sim:/lock/curr_state_r
# code_2
VSIM> examine -binary /sttran/count_v
# 0000000000000000000000000000000011
```

So far we have learned how to set signals with command `force`. However, variables cannot be forced to a new value. If you want to change the value of a variable, you need to use the command `change`:

```
VSIM> change sttran/count_v 12
```

### 3.4.3 Using Breakpoints

Breakpoints are useful when you know approximately (function, procedure, process etc.) where the error is located or you want to know how some code section works. Set a breakpoint in the beginning of the code section of interest and run the simulation. When breakpoint is hit, you can debug the code stepping it line by line.



Step buttons in ModelSim Main/Wave Window Toolbar. Red shows `step` button and green `step over` button. Step command executes the current line of the code and jumps to the next executable line. Step Over command behaves similarly but does not show function and procedure calls.

First you need to enable breakpoints. Breakpoints can be enabled simply by right clicking the line number in Source window and selecting `Enable Breakpoint` from the dialog box. A red dot next to a line number signifies an enabled breakpoint. Note that only executable lines can be set with a breakpoint.

Compare the [state machine diagram](#) to the code. Set a breakpoint e.g. on state `code_2` in file `lock.vhd` and run simulation. For example, line 64 should be reached at approx. 820 ns.

```
VSIM> bp lock.vhd 64
VSIM> run
```

Now examine the values of both incoming `keys_in` bus and the expected constant `third_c`:

```
VSIM> examine -decimal keys_in
# 1
VSIM> examine lock_pkg/third_c
# 6
```



Now continue one step forward:

```
VSIM> step
```

Figure out what went wrong and fix the code.

Compile the modified code. Since we created a `Makefile` for our design in the beginning of this tutorial, you can do it just by calling `make` (from transcript pane or from terminal window):

```
VSIM> make
OR
$ make
```

Now we must verify the functionality of the modified code. Restart the simulation (restart -f); ModelSim loads the modified code automatically when restarted. You can disable breakpoints if you wish (`disablebp`) and then execute the macro (`do lock.do`).

```
VSIM> restart -f
VSIM> disablebp
VSIM> do lock.do
VSIM> run 200
```

As expected, now the design works correctly with that input sequence too.

### 3.4.4 Measuring time

In wave viewer, select from top menus Add -> Cursor. Now you'll see 2 vertical lines (cursors). You can easily drag them and see the time interval between on the bottom. This is very handy when have to measure the execution time (or signal frequency). Just divide the time with the duration of a clock period to derive the number of clock cycles.

### 3.4.5 Using Self-checking testbench

As a last step, compile an automated testbench [tb\\_lock.vhd](#) and execute it:

```
$ vcom vhd/tb_lock.vhd
$ vsim tb_lock &
VSIM> run 10ms
```

You will see error messages if you simulate the original `lock.vhd` whereas the corrected versions passes all the test. Note that testbench halts the simulation when `tb` itself has finished, and the easiest way is to issue an assertion failure on purpose. So you may ignore that "failure".

Create a bug to the lock on purpose, re-compile it, and see if test bench detects the error. Repeat this few times.

## 4. Summary

This concludes our ModelSim tutorial. This was just a brief introduction for how to

get started using ModelSim but there's still a lot more to learn about. For example, commands are often more versatile than what was shown here: breakpoints can, for example, be conditional (i.e. breakpoint can include a condition to determine whether breakpoint is hit or not). A breakpoint can also specify commands that are to be executed when a breakpoint is hit. Check ModelSim Command Reference to see all commands, their options and limitations.

There are also features which were not covered in this document. These include features to measure the `code coverage` (whether or not a statement/branch/condition etc. was executed during a simulation run), a `profiler` (determine CPU/memory usage during a simulation run) and so on. The first type of statistics can be very useful e.g. to measure the goodness of a testbench and the latter can be used to improve simulation run times/memory usage. To learn about these and other features, read ModelSim User Guide.

This is the summary of basic ModelSim commands used in this tutorial.

Design library creation and design compilation commands. Can be prompted in shell or in transcript window.

- `vlib` - Create a new design library
- `vmap` - Define mapping between logical library name and directory path
- `vdir` - List the contents of a design library
- `vdel` - Delete a design unit from a specified library
- `vmake` - Create a Makefile for a specified design library
- `vcom` - Compile VHDL source code into a specified design library
- `vsim` - Invoke the VSIM simulator
- `verror i` - Prints info about warning/error messages, *i* is 4-digit msg number

Simulation commands.

- `view` - Create windows (wave, list, source etc).
- `add` - Add objects to Wave window
- `wave`
- `force` - Force a stimulus to VHDL signals
- `change` - Change the value of a VHDL variable, constant or generic
- `run` - Run simulation
- `restart` - Reload and restart the simulation, option `-f` overrides need for confirmation
- `do` - Execute a macro (do) file

Debug commands.

- `examine` - Examine the value of a signal or a variable
- `bp` - Set a breakpoint
- `disablebp` - Disable breakpoints
- `enablebp` - Enable breakpoints
- `checkpoint` - Save a simulator state
- `restore` - Restore saved simulator state