

Animating Logic Simulations

Project Presentation

Prepared by
Narayanan Raghuraman

Presented on 7th April, 2004

Department of Electrical Engineering
University of Tennessee - Knoxville

ModelSim

- Simulator for VHDL, Verilog and Mixed-language environments
- Delivers High performance with ease of use
- Has inbuilt TCL command interface
- Full featured TCL/TK environment offering script programmability
- **ModelSim + TCL/TK => Innovative visualization tools** to make simulations easier to understand and debug.

Mentor's *ModelSim* HDL simulator has a GUI that is largely written in Tk. The ModelSim command console runs a Tcl/Tk shell, so that you can run your own Tcl scripts from within the simulator. It's also possible to customize the simulator's GUI using Tk. Using TCL, you can access the signals, variables and regions in your design.

Using these features, a custom interface for visualizing the device or a design block can be created. This is discussed in the next slide under 'Extending ModelSim using TCL/TK'

Extending ModelSim with TCL/TK

There are three problem areas for extending ModelSim with TCL to create a custom visualization tool for a device or a board.

- TCL/TK program to generate a simulator to display the target device's output data
- Extract the necessary signal data from the simulation results
- Embedding this extension to the ModelSim GUI to make it behave in an intuitive and convenient way

Suppose you are simulating a device that will drive a small full-graphic display panel, for instance on a mobile phone. Using Tcl/Tk you can create a window displaying the panel's appearance, allowing you to visualise the simulation results directly instead of having to interpret the display controller's output signals.

When creating this kind of extension to ModelSim there are three distinct problem areas.

- A Tcl/Tk program must be written to create the display, based on the simulator's output data. Tcl/Tk's canvas widget is so powerful and easy to use that this task is usually quite straightforward.
- The Tcl/Tk display generator must be provided with the necessary signal data, extracted from the simulation results. ModelSim's "examine" command makes it very easy to get hold of the required data.
- The whole extension must be hooked into the ModelSim GUI environment so that it behaves in an intuitive and convenient way. This means, for example, making the display generator update correctly in response to changes in the position of a cursor in ModelSim's wave window. .

ModelSim - TCL Commands

- When – To execute a function or a set of commands when a signal changes state
- Force – To force the value of a signal to a required value
- Examine – To extract the current value of a named signal
- Wave – Add specific signals to the wave window
- Run – Execute the simulation for a specified time

Example

- Create a clock waveform on signal "clk" with period 40, with value '0' for the first half and '1' for the last half:

```
force clk 0 0, 1 20 -rep 40
```

These four commands are the basic requirements in using TCL to animate and debug VHDL designs using ModelSim. The same can be done with Verilog designs too.

In the example provided, the syntax is described as follows.

Force (signal_name) value1 time1, value2 time2 –rep period

The signal indicated in signal_name is varied between value1 and value2. The period of change is specified after '-rep'. From 'time1' to 'time2' inside the period, the signal is at value1. At 'time2', the signal's value is changed to 'value2'.

Scope of the Project

Tutorial to explain the usage of TCL/TK with ModelSim to create powerful visualization and debugging interfaces

- Part I – Simulation of an Altera UP-2 demo board that demonstrates its user interface features
- Part II – Initiating the student to TCL/TK scripting by providing detailed instructions and making him alter the simulation interface
- Part III – Animation of a Traffic Signal Controller to demonstrate the generation of a more complex visualization and debugging tool

Contribution

A tutorial exploring the usage of TCL/Tk with ModelSim is to be created. The tutorial will consist of three parts. The first two parts will be on animating an Up/Down Counter with TCL. For implementing this, a VHDL based design for the counter will be done as also a TCL based simulation interface. While the first part will give the student a first-hand exposure to animation interfaces with ModelSim, the second part will provide an opportunity to add widgets to the interface and hence learn basic options in TCL/Tk, which are used for animations.

The third part will involve designing a Traffic Light Controller for a Highway/Farm road intersection. This controller will have a few complex options like vehicle sensors and left-turn signals for the highway. The left turn signals will be independent of each other in that activating a left turn signal in only one direction will stop the through traffic in the oncoming direction while not affecting the through traffic in the same direction.

This Traffic Light Controller is designed to give the students an introduction to state machine design and a method to view the results of their design visually. The students will be made to modify the canvas as also the time intervals so that they learn hands on how to design a custom interface.

Deliverables

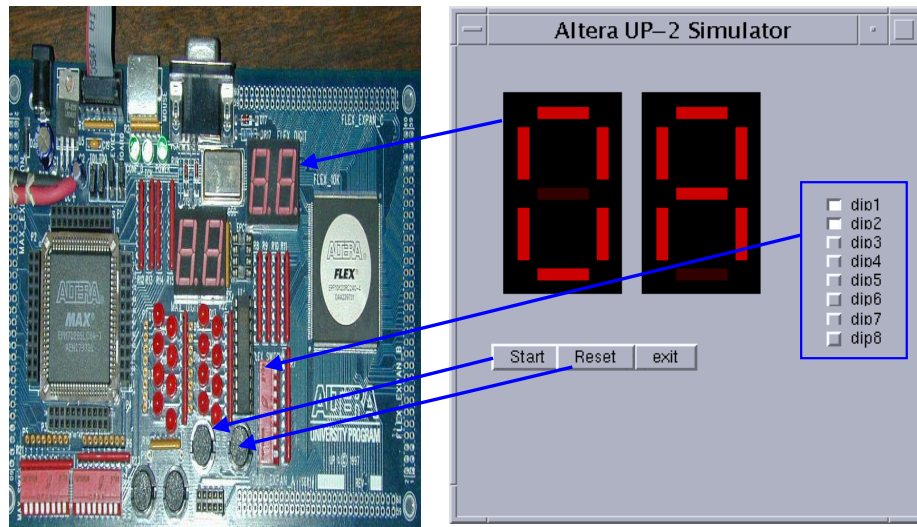
VHDL and TCL files for initial simulation

HTML based tutorial along with steps for compilation and captured gif's

Instructions on how to modify the TCL/Tk source code.

Separate Solutions to the tutorial so that the student is not able to access the solutions before actually performing the steps for modification.

Simulation of Altera UP-2 Demo Board



In the ECE 551 course, the Altera UP-2 Demo board is used to introduce the students to the world of FPGA's. The Altera Demo Board has some user interface features which are

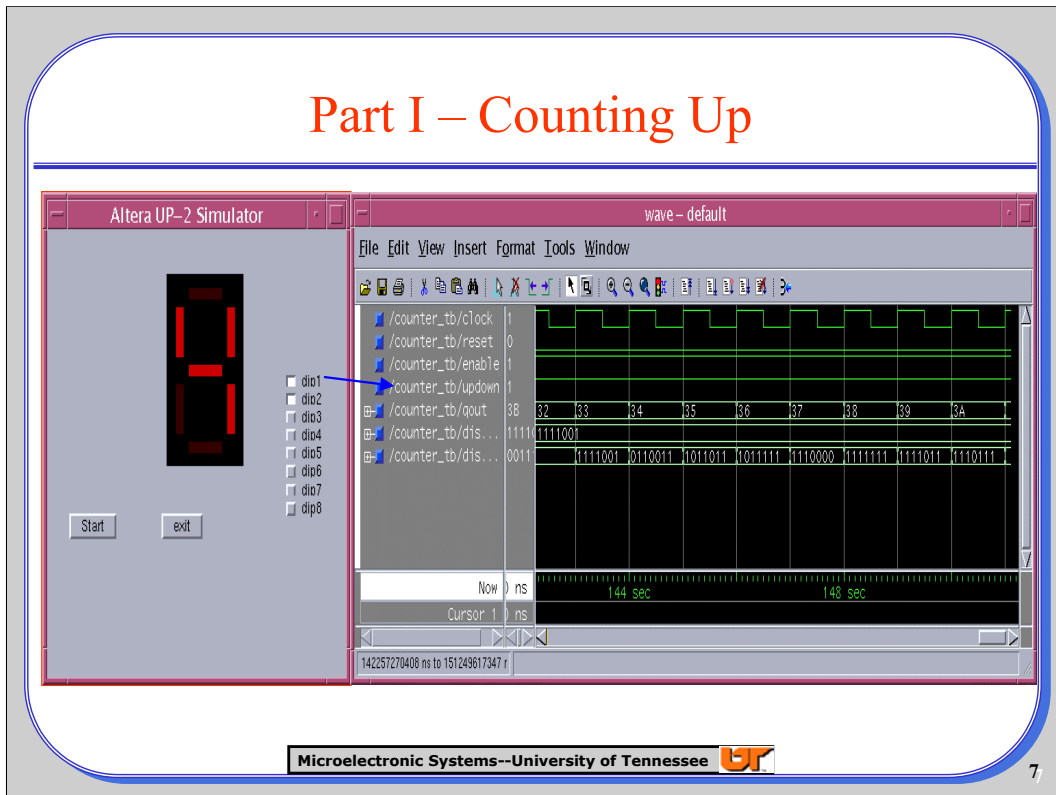
- Two sets of seven segment LED displays
- 8 dip switches
- Two Push buttons

These features are used to simulate the operation of the board. The seven segment displays are simulated by utilizing seven different 'line' objects. Each such 'line' object is connected to one bit of the output signal. The color of the line object is changed when the corresponding bit changes between '1' and '0'.

The dip switches are also used to provide input to the VHDL testbench. Each switch is simulated by a checkbox. A function bound to the checkbox is executed every time it changes state. If it is pressed it provides a value of '1' to the corresponding test bench input.

The two push buttons are named 'Start' and 'Reset' for convenience.

Part I – Counting Up



Before I go on to detail the states of the simulator, I shall explain the TCL/TK Interface a bit.

In TCL/TK, the root window is specified as `.`. All subcomponents of the root window are called widgets.

The canvas is the area of the window where sub-widgets are placed and it can contain signal lights, cars, buttons, radio buttons, check boxes, labels and the slide controls. The canvas normally encompasses the whole window. But we can have sub-canvases. The above mentioned components including the canvas are TCL/TK widgets.

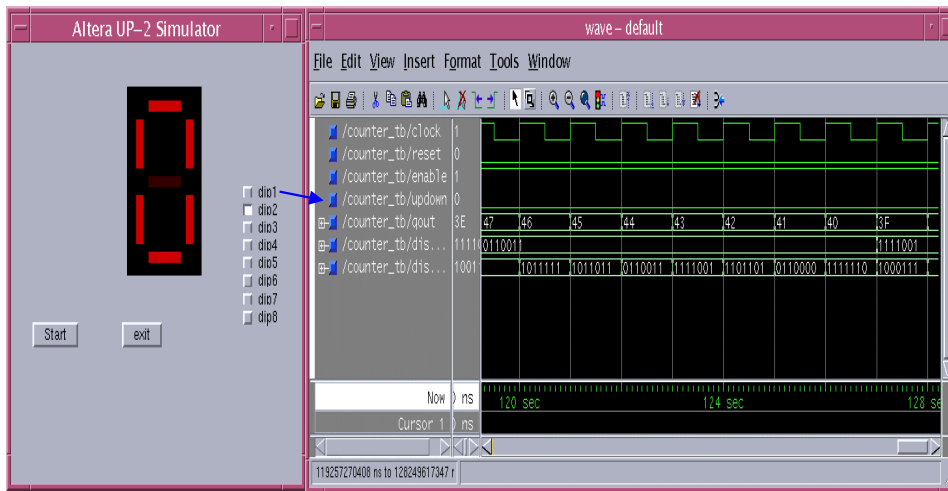
This slide depicts the Simulator in it's counting up state.

Two of the eight dip switches are utilized in this design.

The dip switch 'dip1' is used to set the 'updown' option. The pressed state indicates the up-counting mode. The dip switch 'dip2' sets the 'enable' input of the testbench.

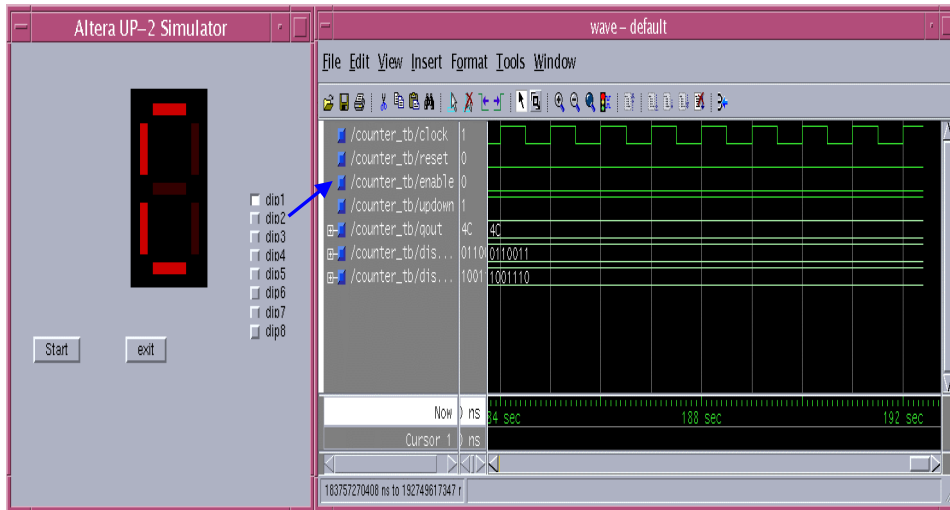
The 'Start' button is pressed to initialize the simulation. The 'start' button is bound with a function

Part I – Counting Down



This slide shows the Simulator counting down. It can be seen the dip switch is placed in its depressed state. Hence it provides a value of '0' as input to the 'updown' signal in the testbench.

Part I – Disabled



In this slide, the dip switch 'dip2' is in its depressed state. Hence the input to the 'enable' signal in the testbench is set to '0'. Hence the Counter stops counting and the value of the lower byte remains at 'C' as shown in the simulation interface as well as the waveform window.

Part II - TCL/TK Interface Modification

STEPS

1. Open the file 'altera.tcl'.
2. Go to the part which says "uncomment the following lines to add 'Reset' Button"
3. Understand the significance of the lines and then uncomment them
4. Next go to the part which says "uncomment the following lines for adding display 2" and uncomment them.
5. Save the file and source the file from 'vsim'
6. The interface should now show the new additions.

This page details the steps to be taken to add a button named 'Reset' and a custom seven segment display widget name 'display_1'.

The complete tcl design has been done in a single file named 'altera.tcl'. This file contains several functional procedures which are used to handle individual functions such as updating seven segment display, sending changes in the checkboxes or buttons to the inputs of the VHDL design and so on.

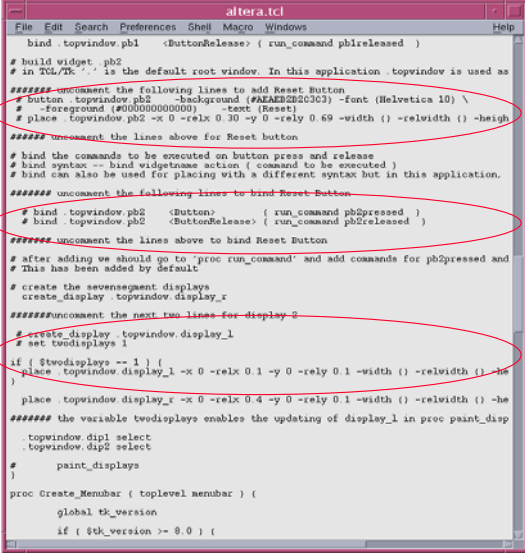
More details about the changes made are provided in the next slide.

How to Modify

Add Reset Button

Bind Reset Button

Add Display2



```
altera.tcl
File Edit Search Preferences Shell Macro Windows Help
bind .topwindow.pb1 <ButtonRelease> { run_command pb1released }
# build widget pb2
# in Tk/Tk8.4 is the default root window. In this application topwindow is used as
##### uncomment the following lines to add Reset Button
# Button .topwindow.pb2 -background {#A9A9A9} -font {Helvetica 10} \
# -foreground {#000000} -text {Reset}
# place .topwindow.pb2 -x 0 -relx 0.20 -y 0 -rely 0.69 -width {} -rely {} -height {}
##### uncomment the lines above for Reset button
# bind the commands to be executed on button press and release
# bind syntax -- bind widgetname action { command to be executed }
# bind can also be used for placing with a different syntax but in this application.
##### uncomment the following lines to bind Reset Button
# bind .topwindow.pb2 <Button> { run_command pb2pressed }
# bind .topwindow.pb2 <ButtonRelease> { run_command pb2released }
##### uncomment the lines above to bind Reset Button
# after adding we should go to 'proc run_command' and add commands for pb2pressed and
# This has been added by default
# create the sevensegment displays
create_display .topwindow.display_r
#####uncomment the next two lines for display 2
# create_display .topwindow.display_l
# set twodisplays 1
if { $twodisplays == 1 } {
place .topwindow.display_l -x 0 -relx 0.1 -y 0 -rely 0.1 -width {} -rely {} -height {}
place .topwindow.display_r -x 0 -relx 0.4 -y 0 -rely 0.1 -width {} -rely {} -height {}
##### the variable twodisplays enables the updating of display_l in proc paint_disp
.topwindow.dipl select
.topwindow.dipr select
paint_displays
}
proc Create_MenuBar ( toplevel menubar ) {
global tk_version
if { $tk_version >= 8.0 } {
```

Before describing the steps, the three steps in the addition of a widget are described.

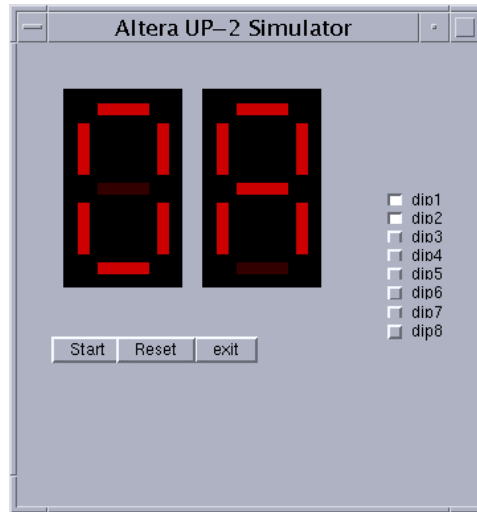
The first step is the **creation** of the widget. Some of the common attributes like the dimensions, the orientation, the background and foreground colors, the text to be displayed on or beside the widget, are set in this step.

The next step is **packing** or **placing** the widgets. The difference between the 'pack' and 'place' command is that the 'place' command can specify the exact location in the canvas. The 'pack' command specifies relative location and also bind the widget to a function. Without this step, the widgets are invisible.

The third optional step in this process is the **binding** of the widgets if necessary. The button widget needs explicit binding if different commands need to be executed at the pressed and released states of the button. For most other widgets the command to be executed is specified in the 'creation' step. These three steps constitute the first change to be made in part II.

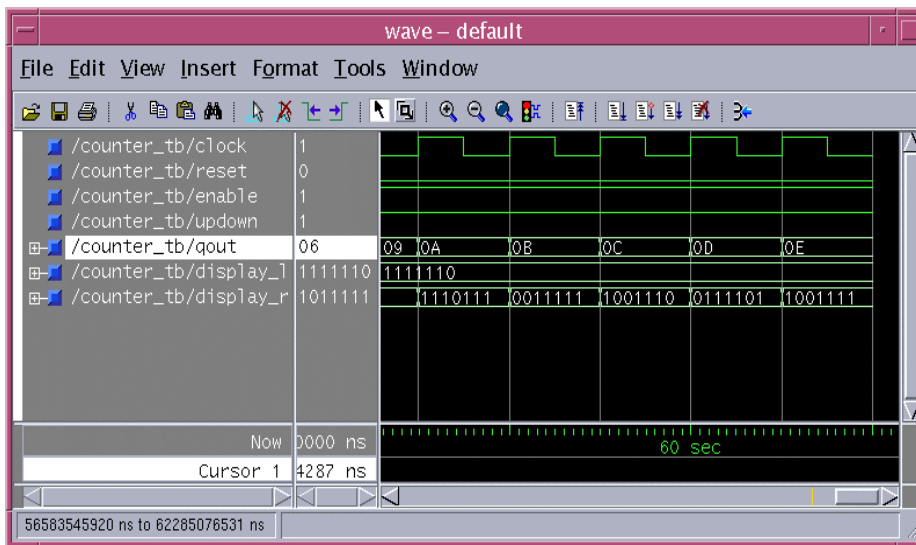
The other change that needs explanation is the addition of the display. This is done with a custom procedure named 'create_display'. A tag is passed as the argument to this function. This tag when passed to the 'update_display' procedure will set the states of the various line widgets to the required state. The two widgets are named as 'display_l' and 'display_r' to signify the left and right display widgets. A variable named 'twodisplays' is used to handle the updating of the display smoothly when the second widget is added. This is used in 'paint_displays' which gets called each time the output changes state.

Part II - Interface



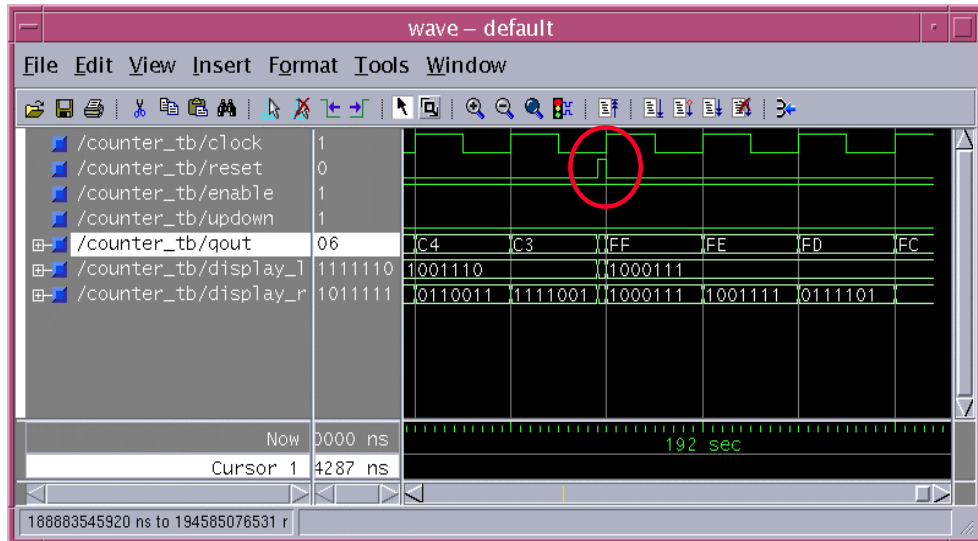
In this slide, it can be seen that the new seven segment display and the new 'Reset' button have been added. The switches dip1 and dip2 are in the pressed state, Hence the counter proceeds in the upward counting state.

Part II – Up Counting Waveform



This waveform shows the waveform for the up-counting state. It can be seen that the output value keeps increasing for every second that passes in the simulation. The value of the counting interval is specified in a functional procedure called 'run_clock'. It is from this procedure that calls the procedure 'update_display' which handles the updating of the displays gets called.

Part II – Reset – Counting Down



In this slide, the effect of 'Reset' on the simulation is shown.

Immediately on pressing the 'Reset' button, the output gets reset to '00'. Since the 'updown' dip switch has been placed in the 'down'('0') state, the counter starts counting down from '00'.

Part III – Traffic Light Controller

- Without creating a device, it might be tough to visualize how the controller will work or debug it quickly.
- Complex VHDL Blocks are also difficult to debug.
- A real life simulation of the device or block will simplify the debugging process.
- A Traffic Signal Controller is simulated to show the benefits of using ModelSim's TCL/TK functionality towards achieving the goals of visualization and debugging.
- The design procedure can be extended to simulate more complex interfaces and VHDL Blocks

Highway Farm-Road Intersection

Inputs to the VHDL Design

- North – South car sensor
- East and West Left turn car sensor
- Green, Yellow and Red state times for the signals

Outputs from the VHDL Design

- East – West , North – South and Turn signals

Internal Signals

- Car waiting times and Current state time

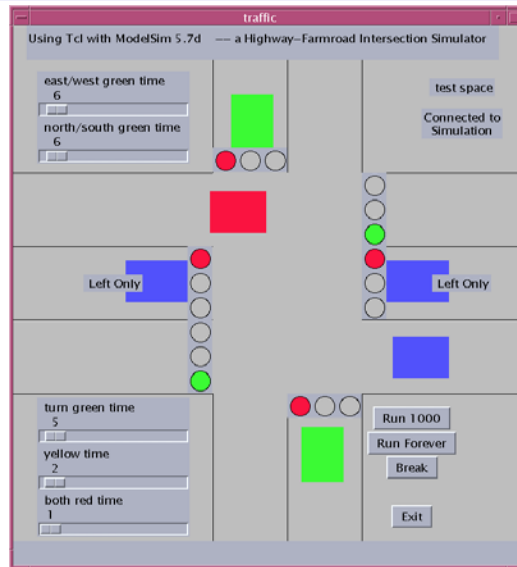
The various input and output signals used in the VHDL Design are shown in this slide. The North – South car sensor and the Turn car sensors are set when a car comes to rest in a particular section of the respective roads.

When the car sensor is activated, the vhdl code's waiting time counter is set into action. This counter is useful in determining which of NS and Turn sensors were activated first. Based on this the state machine changes from EW green to NS or Turn states.

The inputs to the Green, Yellow and Red state times are initialized to a default value. They can also be set using 'scale' widgets which are described later.

The outputs from the VHDL design are used to set the various signals in their green, yellow or red states.

Simulator



This slide shows the simulator that has been designed to depict a Highway – Farm Road Intersection.

- The roads have been created using thin 'line' widgets.
- The cars have been created using thick line widgets with a length of 80 pixels and a width of 20 pixels.
- The signal indicators are created using oval shapes. The different direction signals have ovals placed on separate sub canvasses so that the signal box has an outline.
- Various buttons have been used to control the operation of the simulation.
- The 'scale' widgets are used to set the time for which the signals stay in a particular state.
- A label widget on the top-right corner of the simulator is used for debugging the values of the various flags used to control the simulation. It can also be used for debugging any value in the whole design.

It can be seen that the EW green signal is on and the cars are moving the horizontal direction(EW).

Design Files

VHDL Files

- **Traffic_new.vhd** – Design with State-Machine
- **Tb_traffic.vhd** – Test-bench file which interfaces TCL

TCL Files

- **Intersection.tcl** – Builds the intersection canvas
- **Controls.tcl** – Adds the Timing Controls
- **Cars.tcl** – sets the cars rolling
- **Lights.tcl** – connects the lights to the simulation

These are the various files that have been created for the Simulator/Animator design.

The traffic_new.vhd file has the state-machine design and the assigning of the signals to the output ports according to the different states.

The testbench file provides the mapping to the VHDL file. ModelSim provides an interface wherein the various signals and variables in the loaded design are accessible to TCL scripts sourced in the ModelSim command line.

The TCL files have been programmed as functional procedures.

The functions for building the intersection and the lights(build_intersection in Intersection.tcl) and the controls(light_controls controls.tcl) are executed on startup. The lights are connected to the simulation design by using when statements which are executed by the function connect_lights(lights.tcl).

The file cars.tcl controls the movement of the cars. It has several flags each of which are set inside loops initialized by 'when' statements.

Recursive loops are used to move the cars.

An important point to note here is that the 'when' statements should not be used inside the functions because they are very slow and return only when the signals in the testbench change state. If they are used inside recursive loops, it will make the simulation very slow.

Some Examples of Command Usage

'when'

```
when {/light_ew} { set flag1 [examine light_ew] }
```

'move'

```
$root move $ta [expr -$pos_curr] 0
```

'after'

```
after $tim movecar_ew $root $ta $tim $constr $flag1
```

'create'

```
$root create line $x1 $y1 $x2 $y2 -fill $color -tag $tag -width 60
```

'delete'

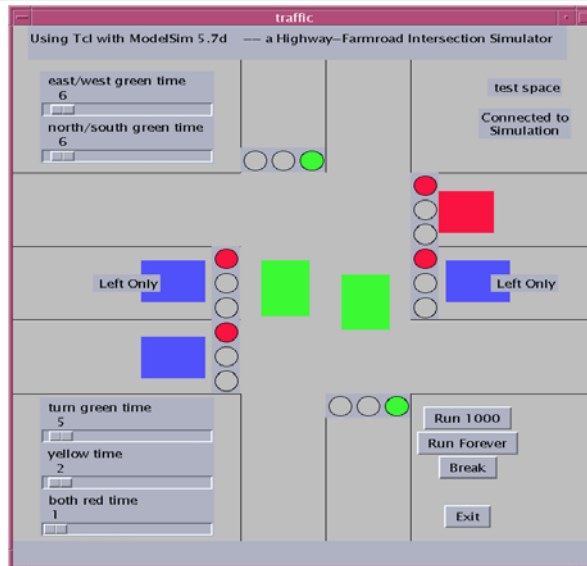
```
$root delete line -tags $ta
```

These are some of the important commands used in this simulation.

1. The 'when' command executes the code inside its braces whenever the state of the variable or signal, to which it is attached, changes.
2. The 'move' command moves the widget pointed to by the tag (\$ta) by a distance specified by the x and y parameters following it. The \$root specifies the root canvas to which the widget belongs.
3. The 'after' command executes its argument procedure or command after the specified time (\$tim) but it returns immediately. It is only the procedure that is executed after the specified time. The other parameters are those that are passed to the functional procedure to be executed.
4. The 'create' command creates a widget. The widget may be any of 'line', 'rectangle', 'oval' and so on. The edge coordinates of the widget are passed as parameters to the 'create' command. \$root is the same as described above and \$tag is the same as \$ta above. The other parameters are self explanatory. More parameters can be added if required.
5. The 'delete' command deletes an already created widget.

Some other commands are 'configure' which can be used to change the attributes of a created widget and 'catch' which can be used to check if a particular widget is available or not.

North – South Green



This slide depicts the Traffic Signal Simulator in its North-South Green state.

The North-South car movement function has been written in such a way that it moves till the NS signal light turns red. If it does become red, then the flag controlling the NS car movement is set to zero, and the NS cars stop at the intersection.

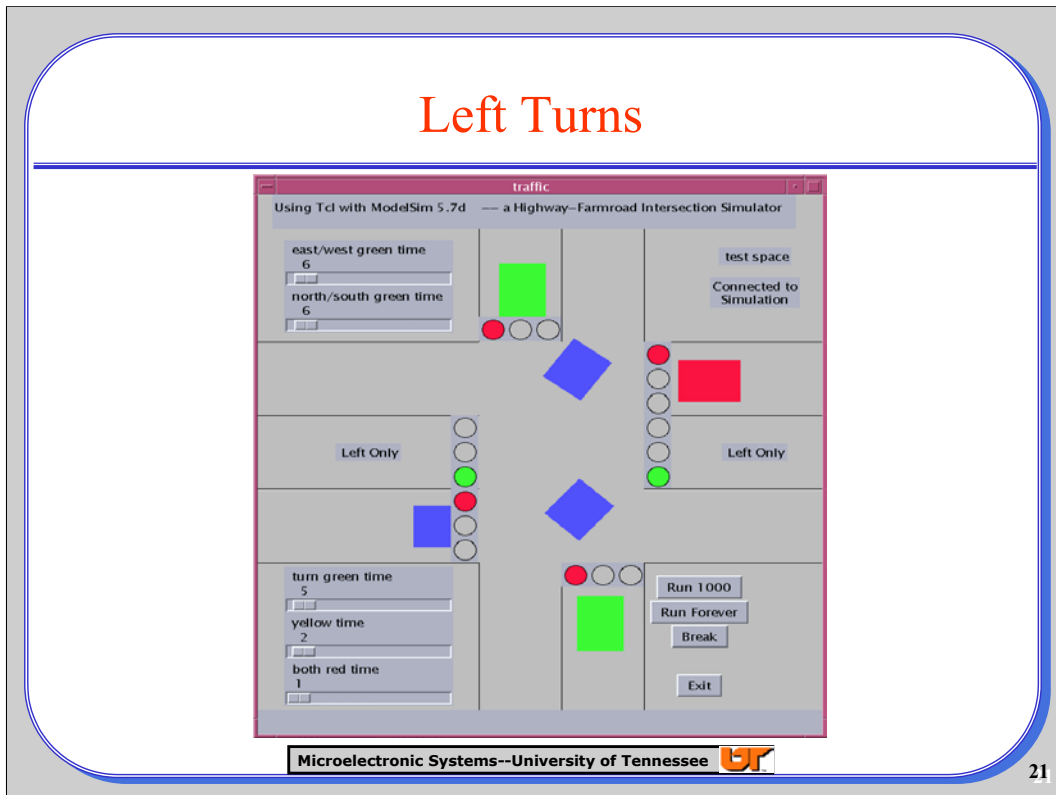
The command 'move' has been used to move the car widget in the vertical direction.

To make the cars stop at the intersection and not anywhere else, the TCL script has been programmed so that it stops only between the particular co-ordinates of the Intersection canvas.

Hence if the Intersection canvas is resized(not the window), then the file cars.tcl has to be altered in such a way that the cars stop in the correct position.

This same logic above applies to the east west lights too.

Left Turns



This slide depicts the Left Turn car movement with the Left turn Signals on.

The Left Turn car movement programming is described here.

Along with that the method of car creation is also described.

The car has been programmed as a 'line' widget with a required length and thickness.

Turns cannot be programmed with the 'move' command because 'move' can operate only in either horizontal or vertical directions.

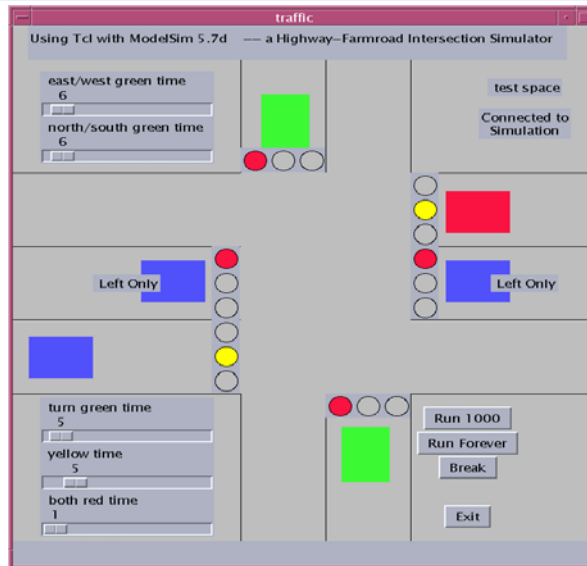
Hence the turn movement has been split into 9 equal steps. Each of these steps has a different widget in a different orientation and location. Here the number 9 has no significance – it is just an arbitrary number.

'when' command checks the state of the Left turn signals as also the other signals. The 'examine' command gets the current value of the signal.

When the left turn signal changes state to green, the turn functions are executed. The Nine different steps are executed in a time staggered manner using the 'after' command.

An important point to note here is that the 'after' command returns immediately and not after the specified amount of time. It is only the function or the command which is called after the specified amount of time.

East – West Yellow



This slide shows the Simulator in the EW yellow state. The simulator has been programmed such that it actually stops car movement when the signal corresponding to the car turns in to the yellow state and the car stays stopped till it becomes green. Thus the simulator follows road traffic regulations !!

Waveform Debug

- The current state of the cursor in the waveform window can be accessed using the ModelSim variable `vsimPriv(acttime)`
- A 'trace' can be placed on this variable so that it calls a function every time this variable changes.
- The WaveTree widget contains all information of the signals in the waveform window.
- Using TCL commands like `get3` and `get4`, the selected signal names and their values can be accessed.
- By combining the operation of `vsimPriv(acttime)` and the WaveTree widget, the Visualization interface can be updated with any change in cursor position.

The visualization interface can be updated by picking up the movements of the cursor, the selected signal names and their respective values from the wave display. To achieve this we need to make extensive access to ModelSim's *WaveTree* widget. The name of the wave tree widget for the primary wave window is commonly `'.wave.tree'`.

This widget has the list of signal names and values that are seen on the left of a wave window. There are numerous Tcl commands giving access to all aspects of this widget.

ModelSim maintains information about its internal state in an array variable named `vsimPriv()`, and `vsimPriv(acttime)` contains the time position of the currently active wave window cursor. We can use Tcl's trace facility to monitor any updates to this variable. When the cursor position changes, the procedure linked to trace will be called and the custom display can be updated inside this procedure.

Examples

`'trace variable vsimPriv(acttime) {rw} callfunction'` - To trace the state of `vsimPriv(acttime)`. The 'rw' signifies that any read or write change will trigger the function named in call function.

`'set selList [.wave.tree curselection]'` - To get the list of signals selected. This list contains the index number of the signals.

`lappend sigList [$win get4 $sig]` - To get the particular signal names, where \$sig contains the number of the signal number

`'set outp [$tree get3 $sig]'` - To get the current value of the signal.

Wave Window Menu Customization

- `Add_menu` – adds a menu in a specified window
- `Add_menu_item` – add a menu item to the specified window and links it to the execution of a function.

Example

```
proc wave_hook {w} { add_menu $w debugt  
  add_menuitem $w debugt traffic [list draw_intersection $w] }
```

- This function can be executed whenever a window is opened by using the user hook variable in ModelSim and including it in 'modelsim.tcl' in the simulation directory.

```
lappend PrefWave(user_hook) wave_hook
```

In order to provide easy access to the new functionality, a new menu item has been added to the wave window. It has been named 'trafficed' to signify 'traffic debug'. This can be done by using the ModelSim commands 'add_menu' and 'add_menuitem'. The function to launch the visualization interface can be linked to the menu item being created. But in order to do this, care must be taken to source the tcl source file.

Another useful option is the `PrefWave(user_hook)` variable. This variable contains a user-specified Tcl list of commands that will be executed whenever any new Wave window is created. The function to create the menu can be appended to this `PrefWave` variable. Similar 'Pref' commands are available for all other windows in ModelSim.

The `PrefWave`, add menu commands and the sourcing of the tcl file can all be included in the `modelsim.tcl` file, which is automatically read by ModelSim at startup.

References

- Mentor Graphics ModelSim Reference Manual
- Performance of Testbenches for VHDL with TCL and C by Johan Karlsson for Nokia
- The Doulos Knowledge website (www.doulos.com)
- Graphical Applications with TCL and TK (2nd edition) by Eric Foster - Johnson

THANK YOU