Columbia University
Department of Electrical Engineering
EE E4340. Tools tutorial.
VHDL design entry and simulation.

There are a few additional set-ups that you will need in your UNIX environment (for those of you in EE E4332, this set-up is the same as yours).

1. Please add the line:

   ```
   . /user/ee4332/public/kshrc
   ```

   to your `.profile` file. This will ensure that you pick up the default environment set up for the class.

2. Please create a file called `.cdsinit`. In this file, please add the line:

   ```
   load("/user/ee4332/public/cdsinit");
   ```

3. Please copy `/user/ee4332/public/cdsenv` to `.cdsenv` in your home directory.

4. Please copy `/user/ee4332/public/cds.lib` to `cds.lib` in your home directory.

5. Please copy the entire directory

   ```
   /usr/cds/tools/dfII/samples/tutorials/composer/tutorial
   ```

   to your home directory:

   ```
   cp -pr /usr/cds/tools/dfII/samples/tutorials/composer/tutorial .
   ```

On-line Cadence documentation can be found by typing `openbook` at the UNIX prompt.

Please do the Cadence Composer tutorial (attached), Chapters 2 - 5. Please use `icfb&` to bring up the Cadence software. It will take several hours to work through the tutorial, but *it is worth it!*. It is very important that you are familiar with Composer, so please don't cut any corners here.

In the remainder of this tutorial, you will learn how to perform VHDL simulation from the Composer interface to Cadence's Leapfrog simulator. Leapfrog is not the easiest VHDL simulator interface to learn, so please also consult the Leapfrog User's Guide in openbook.

In this tutorial, we will enter and simulate a simple logic design in Cadence.

First, create a library to hold your test design. To do this from the CIW, choose **File → New → Library**. You will get a "New Library" form. In this form, enter the name "tutorial2" into the Name field. On the technology file options, choose "Attach to an existing techfile." The Design Manager option should remain "No DM." You will then get an "Attach Design Library to Technology File" window. Choose the library "parts" from the cyclic field and click "Ok." This get the technology information for your library from the "parts" library, which is the technology-specific library that we are using in this course.

In this tutorial, we create a very simple binary counter using a combination of schematics and text VHDL. First create a symbol called "register16." From the symbol editor, choose **Design → Create Cellview → From Pin List**. In the resulting "Cellview from Pin List" form, enter the input pins `reg_in<0:15>` and `clock` and the output pin `reg_out<0:15>`. Choose View Name of "symbol" from the bottom cyclic field. Enter "register16" as the cell name. Hit "OK." "Yes" the hey-box indicating that it is Okay to overwrite the symbol. You can then Okay the "Symbol Generation" form. You may edit the register symbol is you want. Next choose **Edit → Properties → VHDL**. Under "Scalar Data Type," enter `std_ulogic`. Under "Vector Data Type," enter `std_ulogic_vector`. Under "Use Clause," enter `ieee.std_logic_1164.all` and `ieee.std_logic_arith.all`. Under "User Comment," you can add "16-bit register." You can now create the VHDL entity from the symbol. Choose **Design → Create CellView → From CellView**. "OK" the form. The entity that is automatically created now appears in Emacs for you to edit if necessary (actually, this should not be necessary). Emacs is running with some special templates to help the VHDL novice. For more details in this VHDL language-sensitive editor, please see the on-line documentation in "openbook."

Now we need to create the architecture for this 16-bit register. Create a cellview for this cell called "behavior." "behavior" will be the name used for the VHDL architecture. Any name can be used here. On the "Create

New File" form, specify "VHDL-Editor" as the "Tool" field. "OK" the form, which brings up Emacs again. We now need to enter the VHDL architecture for this component:

```
architecture behavior of register16 is
begin
  SYNCH:  process(clock)
  begin
    if (clock = '1' and not(clock'stable)) then
      reg_out(0 to 15) <= reg_in(0 to 15);
    end if;
  end process;
end behavior;
```

Save the design and exit Emacs.

Now, let's create a multiplexer, cell name "multiplexer" with pins `in0<0:15>`, `in1<0:15>`, and `select_in` and output `mux_out<0:15>`. Create the symbol and entity as before. Don't forget to add the correct VHDL properties on the symbol. Now create an architecture in the "behavior" view for this component:

```
architecture behavior of multiplexer is
begin
    process(select_in, in0, in1)
      begin
if (select_in = '0') then
  mux_out <= in0;
elsif (select_in = '1') then
  mux_out <= in1;
else
          mux_out <= "XXXXXXXXXXXXXXXX";
end if;
     end process;
end behavior;
```

The final component we need before assembling the top-level schematic is an incrementer, cell name "incrementer" with pins `incr_in<0:15>` and output `incr_out<0:15>`. Create the symbol and entity as before. The "behavior" view for this component is as follows:

```
architecture behavior of incrementer is
begin
  incr_out <= incr_in + '1';
end behavior;
```

Finally, we need to make the top-level schematic, cell name "top_level." Instantiate a "multiplexer", a "register16", and a "incrementer." There are three inputs of the top-level schematic, `clock`, `start<0:15>`, and `load_start`. Wire up the schematic so that the multiplexer, feeds the register, which feeds the incrementer. The output of the incrementer then feeds back around to go into the `in0` input of the multiplexer. The `in1` input of the multiplexer gets the input from `start` and the `select` of the multiplexer gets the input from `load_start`. Check and save your schematic when you have finished the wiring. Choose **Edit** → **Properties** → **VHDL** and add the VHDL properties in a manner similar to how it was done for the symbols.

We know need to set up the VHDL environment, netlist, and simulate the design. From the CIW, choose **Tools** → **VHDL Toolbox**. We will now create a simple testbench to test this design. Enter the library and cell name ("tutorial2" and "top_level") in the "Top of hierarchy" field. Choose **Commands** → **Create Test Bench**. Leave the "Stimulus File Name" field blank and "OK" the form. The cell name "test" with the architecture "stimulus" is created and displayed in Emacs as a template for your testbench. First modify the signal declarations to include initial conditions:

```
SIGNAL load_start : std_ulogic := '1';
SIGNAL clock : std_ulogic := '0';
SIGNAL start : std_ulogic_vector(0 TO 15) := "0000000000000000";
```

We will enter a very simple testbench into the architecture body after the "dut" instantiation:

```
clock <= not(clock) after 10 ns;
load_start <= '0' after 15 ns;
```

You may also need to add the component binding:

```
FOR ALL: top_level USE ENTITY tutorial2.top_level(schematic);
```

Now the “Top of hierarchy” should be “tutorial2”, “test”, “stimulus.”

Now, we need to set up for netlisting. From the VHDL Toolbox, choose **Setup → Hierarchy**. Select **Switch List** and select “The VHDL Architecture” for the location of the component bindings. “OK” the form. Now choose **Setup → Check** to set up for netlisting. Choose “Export netlists” as “On.” This will write the VHDL netlists to a directory you specify (rather than to /tmp). This can help significantly with debug when you are trying to look at the netlister results to diagnose errors coming from the VHDL analyzer. Specify a directory of your choice as the “Export Directory.” Turn off the “Concatenate Exported Netlists” option. All of the defaults should be fine for the rest of the fields so “OK” the form.

Now choose **Commands → Check Hierarchy** to netlist and syntax check your design, creating a simulatable VHDL model for Leapfrog. If there are any errors, they are reported and Emacs is automatically brought up with the error highlighted. Correct any syntax errors and repeat **Commands → Check Hierarchy** until the design cleanly analyzes and elaborates.

Now, we are going to simulate the design. Choose **Commands → Simulate**. The Leapfrog VHDL simulator window appears. Choose **Tools → Design View**, opening a “Design View” window. Clicking on the top level icon displays the hierarchy under the top level. There should be three components listed: `DUT:TOP_LEVEL`, `$PROCESS_000`, and `$PROCESS_001`. Highlight `$PROCESS_000` and with the right mouse button select **Set Debug Scope**. In the source window, you'll see the source displayed for testbench and the line

```
clock <= not(clock) after 10 ns;
```

highlighted. The designation `$PROCESS_000` corresponds to this line of concurrent VHDL (you can think of each state of concurrent VHDL as representing a process). Double-clicking on `DUT:TOP_LEVEL` causes the hierarchy under this level to expand.

Now we are going to set up for a waveform trace and run the simulator. To trace waveforms, you need to set up a database to hold the waveforms. Choose **Show → Databases**. The “Show” window opens. Choose **Open...**, creating an “Open Database” window. Enter `sim1` into the “Name” field and OK the form. The “SimWave” window should automatically open at this point. Now from the “Design View” window, select `DUT:TOP_LEVEL` and with the right mouse button select **Set Trace → Simple**. This should add all the signals at the top level of the design into the waveform database. They should also appear listed now in the "SimWave" window. To make the names more visible in the waveform window you can choose **Options** → **Customize Window** and select "auto resize names." Now let's run the simulator for 1000 ns. From the VHDL simulation window, choose **Control** → **Set Break** → **Time**. Enter `1000 ns` in the Time field. "OK" the form. Now click the "Run" button on the VHDL Simulator window. Leapfrog should run for 1000 ns and then stop with the waveform recorded into the open SimWave window. To get the waveforms to fit into the window, click the "magnify-equal" icon on the SimWave window.