SYNOPSYS VHDL DESIGN TOOLS

Tutorial
Last Revision 8/23/98

Synopsys is an extensive CAD package for digital design using the VHDL language. Synopsys consists of simulation and synthesis components for both behavioral and structural code. Simulation tools allow you to logically debug the code early in the design process, and then verify timing after synthesis, place and route. Synthesis tools take the VHDL code, convert it to logic, and map that logic to a technology such as Field Programmable Gate Arrays (FPGAs). The goal of this tutorial is to familiarize students with the Synopsys simulation tools.

The tutorial will take the student through the initial setup for the tools. Then into a setup required for each design, a simple example, and an explanation of the debugger and wave form viewer used by the synopsys tools.

I. Setup

The files can be found in my home directory on the CNS Unix systems (such as those in the Riggs 10 lab). My user id is: *fstiver*.

The Synopsys system currently only works on the Solaris operating system. Make sure you are logged into a Solaris machine (most machines in Riggs 10 currently are). Add the following line to your *.cshrc* file at the bottom, disregard the dotted line warning(refer to my *.cshrc* for questions):

```
        source   ~/.synop
```

Copy the *.synop* file from my home directory into your home directory.

```
cp   ~fstiver/.synop   ~<your user id>
```

This change only needs to happen one time. For further designs this step will not need to be repeated. You now need to update your environment. This can be done by either logging out, and logging back in (recommended), or by typing *source ~/.cshrc* in the window that you plan to use.

Next setup the following directory structure from your home directory as follows:

```
mkdir ece426
mkdir ece426/tutorial
mkdir ece426/tutorial/analyze
mkdir ece426/tutorial/counter
```

Next go to your newly created counter directory and copy the following files from my directory *~fstiver/TA426/tutorial/counter*. Use the following commands including the periods:

```
cp   ~ fstiver/TA426/tutorial/counter/.synopsys_vss.setup .
cp   ~ fstiver/TA426/tutorial/counter/count3.vhd .
cp   ~ fstiver/TA426/tutorial/counter/sig.all .
cp   ~ fstiver/TA426/tutorial/counter/testbench.vhd .
cp   ~ fstiver/TA426/tutorial/counter/test.scr .
cp   ~ fstiver/TA426/tutorial/counter/printwindow .
```

Now edit the *.synopsys_vss.setup* file. This file needs to be in each design directory, which is the counter directory and other directories created for future designs.

Go to line 83 which reads:

*ANALYZE : /users/fstiver/**TA426**/tutorial/analyze*

Change it to read:

*ANALYZE : /users/*__userid__*/**ece426**/tutorial/analyze*

With userid being your user name. This informs the synopsys tools where to place all of the analyzed files. Make sure to change the TA426 to ece426 in the line.

Also go to line 128 which reads:

*/users/fstiver/**TA426**/tutorial/counter*

Change it to read:

*/users/*__userid__*/**ece426**/tutorial/counter*

This informs the synopsys tools where the source files for your design are stored. For larger designs this line will be replicated and will include all of the directories in which your parts are stored.

For other design projects simply create another directory off of the ece426 directory and copy the *.synopsys_vss.setup* to that directory. Then edit the two lines above to reflect the correct path to the analyze directory and the path to the source code needed for the design. Also include in this directory the testbench you create to simulate the design.

Note: The parts used to create the design do not need to be contained in the counter directory, but they do need to be included in the *.synopsys_vss.setup* file. For example, the testbench could be in another directory, but that directory would need to contain the *.synopsys_vss.setup* to inform the tools where the source code for each part is contained.

II. VHDL File Structure

The example design presented here is a simple 3 bit counter with an enable. This counter will simply roll over when the counter reaches 111, or 7 depending on which radix the signal is viewed. With each enable the counter counts up one.

Look at the file *count3.vhd*:

```
        Library IEEE;
        use IEEE.STD_LOGIC_1164.all;
        use IEEE.STD_LOGIC_ARITH.all;
        use IEEE.STD_LOGIC_UNSIGNED.all;
```

These four lines inform the tools which standard libraries to use. They are the standard logic library, standard logic arithmetic library, and the standard unsigned logic library. These libraries were installed when the Synopsys tools were installed.

```
entity count3 is
   port( clk     : in   std_logic;
             rst          : in   std_logic;
             enable    : in   std_logic;
             sel          : out std_logic_vector(2 downto 0));
end count3;
```

This is the entity declaration for the counter itself. This defines the connections to the part. It has a global clock(*clk*), and a global reset(*rst*) as inputs. Also there is an enable (*enable*, single bit) as an input and there is a selection line(*sel*) which is a 3 bit output, the actual count.

The remaining lines of the code describe the counter behaviorally. An internal signal *CNT* is used for the outgoing signal *sel*. An output in VHDL cannot be assigned to itself. For example where the line *CNT* <= *CNT* appears, we could not replace it with
sel <= *sel*. The line *sel* <= *CNT* is a continuous assignment statement used to drive the output *sel* to be the exact value of the signal *CNT*. These lines are contained in a process with a sensitivity list which includes *clk* and *rst*. This list is used so that the process will be executed whenever *clk* or *rst* changes. Execution of the process on the change of *rst* will give the counter an initial value of 000. Also execution of the process on the changing of *clk* will make the counter synchronous and each transition of the counter assuming the enable is high will happen on a rising clock edge.

Next look at the *testbench.vhd* file. This file contains the declarations for each part used in the design. The interconnections of the parts and the signals used are declared here also. The clock generator, reset function and testing data are also included.

```
        Library IEEE;
        use IEEE.STD_LOGIC_1164.all;
        use IEEE.STD_LOGIC_ARITH.all;
        use IEEE.STD_LOGIC_UNSIGNED.all;
```

The same libraries are included for the testbench.

```
entity testbench is
end testbench;
```

The entity for the testbench is empty. This is done because the testbench is the top level in the design hierarchy and no other parts will be connecting to it.

The next part of the code is the architecture of the testbench. This contains the signal declarations:

```
architecture test of testbench is

    signal clk          : std_logic:= '0';
    signal rst          : std_logic:= '0';
    signal enable       : std_logic:= '0';
    signal sel          : std_logic_vector(2 downto 0);
```

Next the counter is declared as a component:

```
component count3
    port( clk    : in   std_logic;
                rst        : in   std_logic;
                enable     : in   std_logic;
                sel        : out std_logic_vector(2 downto 0));
end component;
```

Note: This must be exactly the same as the entity declaration for the counter in *count3.vhd*.

After the begin statement the counter is instanced and each of the ports is given a signal name as defined in the declaration.

```
counter   : count3
port map(
            clk    => clk,
            rst    => rst,
        enable      => enable,
        sel    => sel);
```

The name of the part is counter. The name of the component referenced is count3. The next four lines map the ports of the component to the signal name next to it.

Next a generator for the clock must be made.

```
process
begin
   clk <= not(clk);
   wait for 20 ns;
end process;
```

This simply alternates the clock signal from high to low every 20 ns. The clock signal must be given an initial value in the declaration or it will remain in the unknown state for the duration of the simulation. This process will continue during the entire simulation. It does not have a sensitivity list and therefore is executed every time step.

Next a reset function must be made in order to reset all of the parts contained in the design.

```
process
begin
    rst <= '0';
    wait for 53 ns;
    rst <= '1';

wait until(rst'event and rst = '0');
-- stops this process (this is an initial )

end process;
```

This will make *rst = 0* then wait for 53 ns and then make *rst* high. This process will halt at the wait statement. The wait statement will never happen because *rst* will never go back to being low. The process will stop here and wait while other processes continue, which is what we want, reset only needs to happen once.
Note: The double dash(--) indicates comment lines.

The final process is the process to test the counter.

```
process
begin

wait until(rst'event and rst = '1');

countloop:    for i in 1 to 8 loop
```

```
    enable <= '0';
    wait until(clk'event and   clk = '1');
    enable <= '1';
    wait until(clk'event and clk = '1');
end loop countloop;
enable <= '0';
end process;
```

This will wait for the reset to happen and then enable will be driven low. Then wait for the next rising clock edge before changing enable to high. Finally it will wait for another rising clock edge before going to the top of the loop. The last thing the process does will
be to make enable low so as not to leave the counter counting. This is the test harness used to test the part created, in this case it is the counter. The line *wait until(rst'event and rst = '1');* will wait until there is an event on the signal rst(i.e. a change from high to low or low to high) and the value is high before going on. This type of wait statement using the *'event* will cause the statement to occur at an edge of the signal. Please note that in VHDL a sensitivity may not contain any signals if there is a wait statement in the process also.

The last piece of code contained in the testbench is the configuration. Each testbench must have a configuration with it so it can be simulated.

```
configuration c_testbench of testbench is
    for test
    end for;
end c_testbench;
```

The configuration name here is generally the same as the testbench with the addition of the *c_* .
The format is the configuration name followed by the testbench name, and the for statement uses the architecture name of the testbench. The for statement is empty due to the fact that there is only one architecture for the testbench. The single architecture created was behavioral, another architecture that could be created would be structural. Each part can be created either way, but it must be indicated which architecture will be used.

III. Analyzing Files

Now we are ready to start analyzing the files. Each file must be analyzed, and all components that a file uses must be analyzed before that file. So, in this example, we must analyze count3.vhd first and then the testbench. If count3.vhd used a component from another file, such as an adder from add.vhd, we would have to analyze add.vhd, then count3.vhd, and then the testbench.

To analyze the files change into your counter directory off of the tutorial directory. Then type the following command:

vhdlan count3.vhd

Once you have completed compiling the counter code then move onto the testbench code by issuing the following command:

vhdlan testbench.vhd

The final step before simulation is to create a signal trace file. The file you have copied from my directory is *sig.all*. This file allows you to trace any signal in the testbench or in any other parts in the design. The file is formatted as follows:

trace /testbench/'sig*

The format here is the entity of the top level of hierarchy: testbench, followed by **'sig* to look at all of the signals in the testbench.

In order to look at the signals in counter part simply add the following line:

trace /testbench/counter/'sig*

This format is the same as above with the addition of the instance name of the counter, which in this case is counter.

If you would like to view a particular signal such as the *CNT* signal, then instead of using **'sig* with the second line, simply replace it with *CNT*.

trace /testbench/counter/CNT

IV. Simulation

The synopsys tools are equipped with a simulation tool and also a debugging tool. The recommended way to simulate a design is to use the Synopsys VHDL Debugger.

To invoke the debugger, switch to the counter directory which contains the testbench and use the following command:

vhdldbx -i sig.all c_testbench &

The format here is the *vhdldbx* command to invoke the debugger, followed by the *-i sig.all*, which will include the signal trace file followed by the configuration name. The & sign simply makes the command be a background process which enables you to continue to use the window you are currently in.

The Synopsys VHDL Debugger will come up eventually, it may take a while depending on the speed of the machine being used. *Cwaves* is the viewer used by the debugger and will come up after the debugger. It may also be somewhat slow in appearing.

Note: The first time the debugger comes up it will build the fonts needed by the debugger, be patient it will take some time. The debugger will tell you when this is finished. This should only happen the first time the debugger is brought up.

Once the wave viewer is up then go to the debugger window, it should be the blue window. Go to the bottom of the window next to the # sign. To start the simulation simply click in the box with the # sign and type *run 1000*. This will run the simulation for 1000 ns. To run the simulation indefinitely, not recommended, you would just type run, and obviously stop to stop the simulation.

This is the end of the simulation exercise, but more will be explained about the debugger tool in the next section so do not close the tool.

V. Synopsys VHDL Debugger

There are many menus for the debugger. The most used is the Execute menu. This menu contains the options: run, step, next, restart, interrupt and quit. You can use the run option in the menu, but it is easier to specify the time amount to run when typing run at the bottom of the debugger. Also in the middle of the debugger there is a run button that can be used. Simply type in the desired number of ns in the box and then click on the button. This also works like the step function in the execute menu.

Using the Breakpoints menu, a breakpoint may be set on a signal, variable, source line, or a process. This menu is not needed unless there are major problems with the code.

The Monitor and Trace menus are virtually the same. They allow you to monitor or trace a signal, a variable, source line, or a process.

The Stimulus menu may also be used to assign signals or variables specific values. There are options for holding the specified value and also an option to release the value. This can also be done at the command line using the following syntax:

as signal_name '0'   or   as signal_name '1'

The format here is *as* for assign and then the signal name followed by the value. For single bit signals the single quote is used and for multiple bit signals the regular quotes are used as follows(shown here is a 3 bit logic vector):

as signal_name "000"

In the Misc menu there is a cd option which allows the user to move around in the design much like moving around in directories, but using the instance names.

cd counter	This will take the user from the testbench into the count3 code.
cd /testbench	This will take the user back to the testbench.

When using the debugger if a change needs to be made in an instance in the testbench then make the change, reanalyze the part, reanalyze the testbench and the use the restart option on the execute menu. If the testbench needs to be changed then only reanalyze it after the change is made. Then use the restart option on the execute menu.

There is also another option for the lazy typist, a script file can be used and included to run the simulation. Run statements and also assignment statements can be included. Also in larger designs a script file is used to load memory and dump memory.

A copy of a script file is also in my counter directory: *test.scr*, the only line in it now is the run 1000. This type of file does not prove very useful in this example, but it useful later in larger designs. When using the script file a small change needs to be made in the command to invoke the debugger. The change is as follows:

vhdldbx -i sig.all -i test.scr c_testbench &

Note: When changes are made to the script file or signal trace file, no files need to be reanalyzed before another simulation, only a restart on the debugger is needed.

VI. Synopsys Waveform Viewer

The Synopsys Waveform viewer has many useful options also. Waveforms can be saved and loaded using the File menu. Also the tool bar works like a windows program with a small caption coming up when you leave the mouse on it for a few seconds. These buttons include: new window, open waveform, save waveform, cut signal, paste signal, copy signal, binary, octal, decimal, hexadecimal, zoom in, zoom out, zoom fit, previous change, next change, go to start, go to end, place marker at center and refresh. To use the cut, copy, paste, bin, oct, dec, or hex buttons a signal must be selected. Try and change the radix of the selected signal from decimal to hex and then to binary and back.

Printing the waveforms can be done by using a print window function. This function can be found in my counter directory also: printwindow. Simply type printwindow and then click the mouse on the appropriate window.

Using the Edit menu or the buttons, signals can be cut, copied and pasted into different locations in the viewer, above or below another signal. This allows for ease of reading the waveforms.

The Marker menu will allow the user to place a marker at a certain time or in the center of the currently displayed waveform window. It will also allow the user to delete a certain marker. The markers can also be placed at the previous change of a selected signal or the next change of the signal with respect to the marker already in place.

The Go TO menu will allow the user to jump to the start of the waveform or to the end. It will also jump to a specific time, next marker or previous marker.

The View menu has the following options: zoom in, zoom out, zoom fit and full fit. It also has a compress feature which shrinks the size of the signals and a decompress which will do the reverse. It will also let you hide the full name of signals or you can look at the full name of the signals including all of the instance names.

The Options menu allows you to change colors for each of the types of the signals. It also allows for change of fonts. These parameters should not be changed.

VII. Extra Practice

Try using some of the options in menus and using the debugger to become familiar with the tools for use later on in this course.

For some practice try to change the counter from a three bit counter to a four bit counter. Remember that the component, testbench, and instance must all be changed. Also all of the std_logic_vectors for the selection signal must be the same size, as well as the assignment statements must match.

Additionally, there are several other parts that you can use as examples. They can be found in *fstiver/ece426/vhdl_examples* :

var_count.vhd - A generic counter.

umult_shift.vhd - A generic integer multiplier.

full_empty_reg.vhd - A generic register for interfacing between parts. This part is used extensively in the PERL lab for controlling data flow.

testbench.vhd, sig.all, data1.dat, data2.dat - These parts can be used to simulate the umult_shift part. The data is in data1.dat and data2.dat. As detailed above, the part can be simulated by analyzing the umult_shift.vhd, full_empty_reg.vhd and, then, testbench.vhd. The full_empty_reg part is used to interface the part for input and output in this simulation. It could be used in future applications to interface the part to other parts. The testbench.vhd part must be analyzed last since the other 2 parts are used inside testbench.vhd.

mult_pipe12x36.vhd - Performs the integer multiplication of a 12 bit and a 36 bit value, producing a 48 bit result. It is pipelined to produce a result every 2 cycles once the pipeline is full.

Generic means in the above cases that the parts can accept whatever size input/output data length you wish. You set the width in the component declaration.

As you become more comfortable with VHDL and simulation, you should attempt to simulate the above parts. The testbench is already set up for the umult_shift part. The design of the state machines and coding style may prove useful.