University of California, Davis
Department of Electrical and Computer Engineering

EEC180B Spring 2006

LAB 1: ModelSim/Synplicity/Xilinx Tutorial and Lab Exercise

Objective: This tutorial covers the complete design flow for implementing a high-level VHDL design in a Xilinx FPGA. We provide a VHDL description of a simple 16-bit up/down counter to illustrate the design procedures. The first step is to verify the VHDL model through functional (behavioral) simulation using ModelSim. After the design has been successfully simulated, you will synthesize the design into hardware using Synplicity's Amplify tool. You will then use the Xilinx ISE software to map the design into a Xilinx Spartan-3 FPGA device and download the resulting programming file to a Spartan-3-based board for verification.

After completing the up/down counter implementation, you should be able to implement similar designs. Thus, at the end of the tutorial, an exercise is assigned to test your understanding of the high-level design process.

Running ModelSim

1. To configure your environmental variables so that you can run ModelSim, run the setup script as shown below:

   ```
   setup modelsim61
   ```

   Another option is to manually edit your .software file in your home directory and add the line "modelsim61" at the bottom (without quotes). For example, your .software file might look like the following after you are done:

   ```
   @standard
   modelsim61
   ```

   Do *not* remove the line with @standard. (Note – the setup script just has to be run **once** to modify your .software file. Once you have run the setup script and modified your .software file, your environmental variables will be setup correctly each time you log on or open a new window.)

2. Presumably you will generally be working on the HP workstations in 2107. However, ModelSim, Synplicity and Xilinx ISE will not run under HP-UX. Therefore, you will need to open an X-window session on a Linux workstation. From an HP workstation in 2107, open a new window on a Linux workstation using the following command:

   ```
   rxt ugrad-linux
   ```

   This will open an X-term window on the least-loaded machine in 1101 and 1105. (You should not be logged on to the same Linux machine as any of your neighbors or it will be very slow!). Another option is to open a window on a Linux machine in 2110 (intel01-intel20) using the rxt command. However, if a student is running Quartus on the console, your programs will be unbearably slow. The command "**top**" is useful to see what other processes are running on the machine you are on. To exit "top", hold down the Ctrl key and type C (Ctrl-C).

3. Create a project directory and copy the files from /afs/ece/classes/eec180b/lab1 to your project directory.

```
owl<21> cd eec180b
owl<22> mkdir lab1
owl<23> cp /afs/ece/classes/eec180b/lab1/* ./lab1
owl<24> cd lab1
owl<25> ls
my_pkg.vhd  tb.vhd  updown.vhd
owl<26>
```

   These actions have created a project directory ~/eec180b/lab1 and copied the VHDL source files into it. Take a look at the VHDL source files. You will be simulating a simple updown counter which displays the binary count in hexadecimal on four seven-segment display modules. The updown.vhd and my_pkg.vhd files implement the counter and the display logic, while the tb.vhd file is the simulation testbench and will not be synthesized. (See the Appendix for an explanation of the source code.)

4. Run ModelSim using the following command:

   vsim &

   You can close the "Welcome to ModelSim window". If you want to open it again later, you can find it under the Help menu.

5. Select **File>New>Project** to create a new project.
   Set Project Name to: **updown**
   Set Project Location to your project directory **(/home/*username*/eec180b/lab1**)
   Set Default Library Name to: **work**
   Click **OK**
   (If you are given a choice about which modelsim.ini file to use, choose the default.)

6. A window titled "Add Items to the Project" will pop up.
   Double-click the **Add Existing File** icon.
   Click the Browse button and select the three VHDL source files in your project directory. Since these are already in your project directory, you should click the button: **Reference from current location**
   Click **OK**
   Close the "Add Items to the Project" window.

7. Select **Compile>Compile Order** to arrange the compile order of the files. Based on the dependencies (my_pkg.vhd is referenced in updown.vhd and updown.vhd is referenced in tb.vhd), arrange the compile order as
   my_pkg.vhd
   updown.vhd
   tb.vhd

8. Select **Compile>Compile All** to compile the files. You should not get any errors.

9. Select the **Library** tab at the bottom of the Project window.
   Expand the work directory by clicking the "+" box. (It may take a minute or two until the object files are written into the work library.)
   Double-click the **tb (Entity)** to load the design unit.

10. Select **View>Debug Windows>Wave** to open the wave window for simulation output. The Objects window should already be open. If not, open the Objects window by selecting **View>Debug Windows>Objects**.

11. Click on the Objects window and then select **Add>Wave>Signals in Region** to copy the testbench signals into the Wave window.

12. From the Workspace pane of the ModelSim window, make sure the "**sim**" tab at the bottom of the window is selected and then click on **uut**. Uut is the instance name of the updown module and means "Unit Under Test". The signals in the updown module should now be displayed in the Objects window.

13. From the Objects window, select the **count**, **clkcount** and **clkout** signals by clicking on each one with the Shift key held down. Select **Add>Wave>Selected Signals** to add these signals to the others in the Wave window. Verify that **testmode**, a signal, has the value 1 in the Objects window.

14. From the Workspace pane, select **tb** again so that the top-level signals are in the Objects window. Since the simulation testbench does not drive the **enable** or the **up_dn** input signals, we will control these interactively from the Objects window.

15. Set both **enable** and **up_dn** to 1 in the Objects window as follows:
    Select **enable**, right-click and then select **Force…** and change the value from U to 1. (Kind can be left on "Freeze"). Do the same for up_dn:
    Select **up_dn**, right click and then select **Force…** and change the value from U to 1.

16. From the Wave window, change the radix of all the signals to hexadecimal.
    Select **Edit>Select All**, then select **Format>Radix>Hexadecimal**.

17. At the VSIM promt in the ModelSim window, type **run 4000 ns**. You should see the first few cycles of **clkout** in the wave Window. (To view the Wave window easier, it

can be helpful to "undock" it by clicking the button in the upper right of the window between the "**+**" and the "**x**" buttons.)

Note that for each full cycle of **clkout**, each of the **AN0**-**AN3** mux signals is pulsed low once, displaying the count value in hex on the four-digit display. Study the VHDL files so that you understand how the updown counter works. (An explanation of the updown.vhd file is provided in the Appendix.)

18. Simulate until the **count** value reaches at least 0003 and then force **up_dn** to 0 and run until the **count** value rolls over from 0000 to FFFF.

19. Force **enable** to 0 and simulate for several **clkout** cycles. The **count** value should not change.

20. Print the wave window to a Postscript file (i.e. wave.ps) and print it. (Another option is to convert the file to pdf using the ps2pdf utility on either a Linux or HP workstation and then print the pdf.)

21. **<u>Demonstrate your simulation to your TA</u>. (25 points)**

<u>Running Synplicity synthesis</u>

1. To setup your environment for the Synplicity tools, run the following command:

   setup synplicity

   The setup script will insert a line in your .software file, which is in your home directory. After running the setup script, you must open a new window.

2. From your project directory, run Synplicity's amplify tool by typing the following command:

```
amplify &
```

3. Open a new Project by clicking either the **P** icon in the upper-left corner or on the **Open Project** button. Click on the **Project Wizard** button.

4. Fill in the New Project information as follows:
   Select Project Type: **Synthesis**
   Project Name: **updown**
   Project Directory: Use the "**…**" button to browse to your project directory. It should be something like **/home/*username*/eec180b/lab1** (or whatever you named your project directory).
   Project File: This will be named for you based on the Project Name and Directory.
   Click **Next>**

5. Click the **Add File…**button and add the synthesizable VHDL files, **my_pkg.vhd** and **updown.vhd**, from your project directory to your Project Files. (Note that the tb.vhd file is not synthesized – it is only used for simulation.) After adding the files, click **OK**. Then click **Next>**

6. On the next page of the Project Wizard, select the following:
   Technology: **Xilinx Spartan3**
   Part: **XC3S200**
   Speed: **-4**
   Package: **FT256**
   Click **Next>**

7. Accept the Project Wizard-Options page defaults (FSM Compiler and Resource Sharing) and click **Next>**

   Accept the Project Wizard-Constraints page defaults and click **Next>**

   Accempt the Project Wizard-VHDL Options page defaults. Specifying the top level entity name is optional so you can leave that box blank. Click **Finish>**

8. Click the **Impl Options…** button. Under the Implementation Results tab, check that the following are specified by default.

Implementation Name:	**rev_1**
Results Directory:	**/home/*username*/eec180b/lab1/rev_1**
Result File Name:	**updown.edf**
Result Format:	**edif**

   The button for Write Vendor Constraint File should be selected and then click OK. You can change the Implementation name to **updown** or some other name if you desire. Click **OK**

9. Click the **Run** button to synthesize the design. There shouldn't be any errors. Check the warnings and notes by selecting the appropriate tab at the bottom of the window. You can ignore the warnings about "Others clause is not synthesized" in the my_pkg.vhd and updown.vhd files. The "others" condition will never occur in either of these files.

10. Select **File>Exit** and answer **Yes** to the question, "Do you want to save the changes to your project?" Note that there are *two* important output files from the synthesis tool: **updown.edf**, which is the synthesized output file, and **updown.ncf**, which contains the pin location constraints for the Xilinx place-and-route tool. These should both be in your Implementation Results directory (rev_1 or whatever you named it). In contrast, the project file, **updown.prj**, will be stored one level above this in the project directory.

Running Xilinx ISE

1. Set up your environment for the Xilinx ISE software by running the setup script:

   ```
   setup ise71
   ```

   Open a new window on a Linux machine and cd into your project directory.

2. Start the Xilinx ISE program by typing the following command:

   ```
   ise &
   ```

3. Open a project by selecting **File>New Project…**. Specify the project as follows:

Project Name:	**rev_1**
Project Location:	**/home/*username*/eec180b/lab1/rev_1**
Top-Level Module Type:	**EDIF**

   Then click **NEXT >**
   [Note that the Project Location **must** be the directory where the **updown.edf** and **updown.ncf** files generated by Amplify are located.]

4. On the next page, select the Input File for the Project to be: **updown.edf**
   You can leave the Constraint File box empty since the updown.ncf file will be used automatically. (You can also specify it if you want.)
   Click NEXT > for the remaining dialog boxes and then click FINISH.

5. Select **updown.edf** in the Sources window (upper-left). In the lower-left window, expand Generate Programming File by clicking the "+" box next to it. Then select **Programming File Generation Report**, right-click and select Run. When the processing is done, verify that there are no errors. You should have green checkmarks next to the Implement Design, Generate Programming File, and Programming File Generation Report processes.

6. Check any Warnings using the tab at the bottom of the window. It is also a good idea to open the Pad Report, which can be accessed via a link at the bottom of the Design Summary window or by expanding the Implement Design and Place & Route processes and double-clicking on the Pad Report. Make *sure* that the pad locations specified in **updown.ncf** were used in the pad assignments by checking the pad assignments against the constraints in the updown.vhd source file. If these are not the same, you may have used a project directory that did not contain the updown.ncf file. The programming file that you will use to program the Xilinx Spartan-3 device is **updown.bit**.

7. Close the ISE program.

Downloading to the Xilinx Spartan-3 board

The procedure for downloading the programming file to the Spartan-3 board is as follows:

1. Set up the Xilinx board. Connect the download cable from the PC's parallel port to the JTAG header (J7) on the Xilinx board. **Make sure that the signals at the end of the JTAG cable align with the labels listed on the board as shown in Figure 1 below**. Plug the AC wall adapter into a power outlet and connect the barrel connector to the Xilinx board. There is no power switch on the board so the board will be powered whenever the power cable is plugged in to both the board and the outlet.

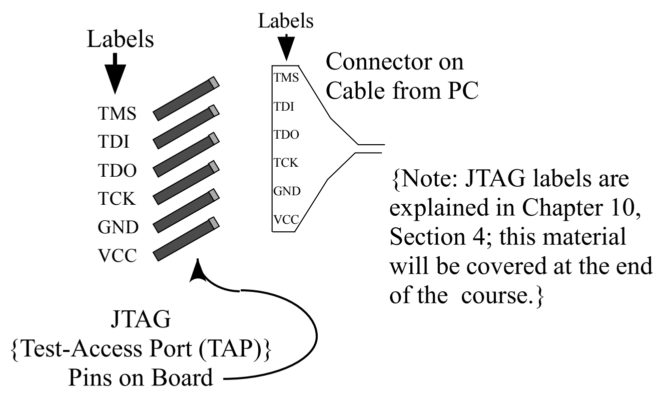
Labels
TMS
TDI
TDO
TCK
GND
VCC
Labels
TMS
TDI
TDO
TCK
GND
VCC
Connector on Cable from PC
{Note: JTAG labels are explained in Chapter 10, Section 4; this material will be covered at the end of the course.}
JTAG {Test-Access Port (TAP)} Pins on Board

Figure 1. JTAG Connector Alignment

2. Log-in to the Linux or Windows PC that will be used as the download station.

3. Run the Xilinx ISE program

```
ise &
```

4. Select **File>Open Project…** and select the **updown.ise** project file in your Implementation Results directory (rev_1, unless you used an alternate name).

5. From the lower-left window, select **Configure Device (IMPACT)** under **Generate Programming File**. Right-click and select **Open Without Updating** since you have already run the place-and-route program on your Linux workstation.

On page 1, select Boundary Scan Mode (default) and Next>
On page 2, select Automatically connect to cable and identify Boundary Scan chain and then click Finish.

For the xc3s200, select the updown.bit file as the configuration file and click Open. Ignore the Warning about the Startup Clock and click OK. For the second device (xcf02s), just select the BYPASS button in the lower-right corner instead of a configuration file.

6. Select the xc3s200 device, right-click and select **Program…**. Click on OK without adding any extra options. The bit file should be downloaded to the board and you should see the message "**Programming Succeeded"**. Now you can test the updown counter using the SW7 (enable) and SW6 (up_dn) inputs.

7. Remember to log out when you are done so that others don't gain access to the files in your workstation account.

Exercise (50 points)

Modify the counter example so that it works like a timer displaying minutes and seconds. The lower two digits should represent seconds and should count from 00 to 59 (in decimal!) and then start at 00 again. The upper two digits should represent minutes and should also count from 00 to 59 in decimal. You don't need to adjust the clock divider to make the time-keeping accurate (although you can if you want!). You can still use the 3 Hz rate to increment the seconds counter. You will not need the enable or up_dn control signals for the timer since it will always count up and cannot be disabled. However, reset should set the timer back to 0000.

Hint: Define four 4-bit signals (std_logic_vectors) for the four digits and use if-then statements and relational operators such as = and /= to specify the logic for your timer. Study the COUNTER process for examples of various if statements. For example, to check if a 4-bit std_logic_vector named LS_sec is not equal to 9, you could use the statement:

```
                if (LS_sec /= "1001") then
```

Simulate, synthesize and download your new design.
Demonstrate to your TA that your design works on the Xilinx board.

Lab Report (25 points)

Submit the following items for verification and grading:

- VHDL source code for your timer design.
- Simulation traces of your timer design.
- Verification sheet signed by your TA on successful testing of your timer circuit.

Appendix A – VHDL source files

my_pkg.vhd

```
-- Package for updown counter example
-- The function hex_7seg takes a 4-bit hex input and produces
-- the corresponding 7-segment display driver signals, which
-- are active low.  The segs (0 to 6) correspond to segments
-- A, B, C, D, E, F, G, respectively.

Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

package my_pkg is
subtype hex is std_logic_vector(3 downto 0);
subtype segsAtoG is std_logic_vector(0 to 6);
constant zero : std_logic_vector(15 downto 0) := "0000000000000000";
function hex_7seg (hexval : hex) return segsAtoG;
end my_pkg;

package body my_pkg is
function hex_7seg (hexval : hex) return segsAtoG is
begin
   case hexval is
      when "0000" => return ("0000001");
      when "0001" => return ("1001111");
      when "0010" => return ("0010010");
      when "0011" => return ("0000110");
      when "0100" => return ("1001100");
      when "0101" => return ("0100100");
      when "0110" => return ("0100000");
      when "0111" => return ("0001111");
      when "1000" => return ("0000000");
      when "1001" => return ("0001100");
      when "1010" => return ("0001000");
      when "1011" => return ("1100000");
      when "1100" => return ("0110001");
      when "1101" => return ("1000010");
      when "1110" => return ("0110000");
      when "1111" => return ("0111000");
      when others => return ("-------");
   end case;
end; -- function
end my_pkg;
```

updown.vhd

```
-- Updown counter example.
-- Displays binary count on 4-digit hexadecimal display.
-- Counts up/down in range of 0000 to FFFF.

Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use WORK.my_pkg.all;

entity updown is
   generic (test_mode : integer := 0);
   port (clk, reset, enable, up_dn : in std_logic;
        AN3, AN2, AN1, AN0 : out std_logic;
          disp : out segsAtoG);

attribute xc_loc : string;
attribute xc_loc of clk: signal is "T9";
attribute xc_loc of reset: signal is "L14";
attribute xc_loc of enable: signal is "K13";
attribute xc_loc of up_dn: signal is "K14";
attribute xc_loc of AN3: signal is "E13";
attribute xc_loc of AN2: signal is "F14";
attribute xc_loc of AN1: signal is "G14";
attribute xc_loc of AN0: signal is "D14";
attribute xc_loc of disp: signal is "E14,G13,N15,P15,R16,F13,N16";
end updown;

architecture behave of updown is
signal count : std_logic_vector(15 downto 0);
signal clkcount : std_logic_vector(23 downto 0);
signal mux_disp : std_logic_vector(3 downto 0);
signal clkout : std_logic;
begin

-- use divided clock for synthesis, but not simulation
clkout <= clkcount(23) when (test_mode=0) else clkcount(3);
mux_disp <= clkcount(10 downto 7) when (test_mode=0) else
               clkcount(3 downto 0);

DISPLAY : process(mux_disp, count)
begin
   AN0 <= '1'; AN1 <= '1'; AN2 <= '1'; AN3 <= '1';    -- defaults
   disp <= "1111111";                                 -- defaults
   case mux_disp is
      when "0000" =>           disp <= hex_7seg(count(3 downto 0));
      when "0001" | "0010" =>  AN0 <= '0';
                               disp <= hex_7seg(count(3 downto 0));
      when "0011" =>           disp <= hex_7seg(count(3 downto 0));

      when "0100" =>           disp <= hex_7seg(count(7 downto 4));
      when "0101" | "0110" =>  AN1 <= '0';
                               disp <= hex_7seg(count(7 downto 4));
      when "0111" =>           disp <= hex_7seg(count(7 downto 4));

      when "1000" =>           disp <= hex_7seg(count(11 downto 8));
      when "1001" | "1010" =>   AN2 <= '0';
                                disp <= hex_7seg(count(11 downto 8));
      when "1011" =>            disp <= hex_7seg(count(11 downto 8));

      when "1100" =>            disp <= hex_7seg(count(15 downto 12));
      when "1101" | "1110" =>   AN3 <= '0';
                                disp <= hex_7seg(count(15 downto 12));
      when "1111" =>            disp <= hex_7seg(count(15 downto 12));
      when others =>    null;
   end case;
end process;

CLOCK : process            -- divide 50 MHz oscillator by 2**24
begin
wait until clk'event and clk='1';
if (reset='1') then        -- synchronous reset (active-high)
  clkcount <= "000000000000000000000000";
else
  clkcount <= clkcount+1;
end if;
end process;

COUNTER : process(reset, clkout, enable, up_dn, count)
begin
if (reset='1') then        -- reset active high
   count <= zero;          -- asynchronous reset
elsif clkout'event and clkout='1' then
   if (enable='1') then
      if (up_dn='1') then
               count <= count+1;
      else
               count <= count-1;
      end if;
   end if;
end if;
end process;

end behave; -- architecture

configuration updown_cfg of updown is
for behave
end for;
end updown_cfg;
```

<u>Explanation of updown.vhd code:</u>
The updown.vhd code is not trivial, so some detailed explanation will be provided here in an attempt to document and clarify the design.

1. The test_mode generic, located inside the entity part of updown.vhd, is used allow a clock divider to be implemented during synthesis, but bypassed during simulation. The clock signal provided the Xilinx Spartan-3 board is a 50 MHz signal. However, we do not want to run our updown counter at that rate because we would not be able to read the output display. Therefore, we implement a 24-bit binary counter, clkcount, which is implemented in the CLOCK process. For synthesis, the msb of clkcount, clkcount(23), is used as the clock signal, clkout, for the updown counter. Thus, the frequency of the updown counter is 50 MHz / 2**24 ≈ 3 Hz. (Verify that this is correct.)

   For simulation, however, we do not want to simulate 2**24 clock cycles just to produce one clock cycle for the updown counter!! Thus, for simulation we set test_mode=1 and bypass the clock divider. For simulation we use clkcount(3) as the clkout signal that clocks the updown counter so that only 2**4 = 16 cycles are required for a clkout cycle. (We will explain why clkcount(3) is chosen as the simulation clock signal later.)

2. Within the entity, the attribute statements are used to assign pad locations to the port signals. For example, pad K13 is used for the enable signal. These statements are used by Synplicity's Amplify to generate a constaint file (.ncf) for the Xilinx place-and-route tool. These pin assignments can be found in the <u>Spartan-3 Start Kit Board User Guide</u>, which is posted on the course web site.

3. The basic code for the updown counter is found in the COUNTER process. This code implements a simple updown counter with enable and up/down control signals and an asynchronous reset.

4. Most of the complexity of the updown.vhd code is due to implementation of the four-digit, seven-segment LED display on the Xilinx board. Each of the four digits shares the eight common control signals, A-G and DP (decimal point), to light the individual LED segments. Each of the four digits also has a separate anode control signal which enables or disables the digit. Thus, only one of these control signals, AN3-AN0, should be low at any time to enable just one digit at a time. However, by switching quickly between the four digits, the user perceives that all four digits are lit simultaneously. (See Chapter 3 of the <u>Spartan-3 Starter Kit Board User Guide</u> for a more detailed explanation.)

   The DISPLAY process is used to implement the "scanning" technique (described above) of the output display. Four bits of the clkcount binary counter are used to cycle through the four digits. For example, the following code outputs the least significant hex digit of count to the least significant seven-segment display. (Note: hex_7seg() is a function defined in my_pkg.vhd.)

```
case mux_disp is
   when "0000" =>           disp <= hex_7seg(count(3 downto 0));
   when "0001" | "0010" =>  AN0 <= '0';
                            disp <= hex_7seg(count(3 downto 0));
   when "0011" =>           disp <= hex_7seg(count(3 downto 0));
```

First, when mux_disp is “0000”, the seven-segment driver signals for the least significant digit are assigned to the common control signals, A-G and DP. Then when mux_disp is “0001” and “0010”, the anode control signal AN0 is set low to activate the least significant seven-segment display digit. Finally when mux_disp is “0011”, AN0 is set high again to disable the least significant digit. Thus, the driver control signals will be stable for one count before AN0 goes low and be held stable for one count after AN0 goes high again. In essence, the driver control signals have a one count “set-up time” and a one count ‘hold time”, which should prevent glitching on the control lines when AN0 is low.

5. Mux_disp is set to clkcount(10 downto 7) for synthesis and clkcount(3 downto 0) for simulation by using the test_mode generic. In the synthesized design, clkcount increments at a rate of 50 MHz. Therefore, the frequency of clkcount(10) will be 50 MHz / 2**11 ≈ 24.4 kHz. The count value is updated at a rate of approximately 3 Hz, so the four-digit display will be scanned about 2**24/2**11 = 2**13 = 8192 times per count value. This fast scanning rate is sufficient to appear as a continuous, simultaneous display of all four digits. Of course, for simulation, we want to eliminate simulation of clock divider circuits as much as possible so we use the lowest four bits of clkcount to control the output display scanning. It takes 16 cycles minimum to implement the four-digit scanning with the one-cycle setup and hold times as described above. Therefore, count should stay constant for 16 clock cycles so that each value of count can be displayed in simulation. For this reason, clkout in simulation is specified as clkcount(3).

tb.vhd

```
-- testbench for updown counter example
-- Generates the clk signal using a process.
-- Initializes the reset signal using a process.
-- up_dn and enable are not set. These can be controlled
-- manually during simulation.
-- test_mode generic set to 1 to bypass clock divider

Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use WORK.my_pkg.all;

entity tb is
end tb;

architecture behave of tb is
signal clk, reset, enable, up_dn, AN3, AN2, AN1, AN0 : std_logic;
signal disp : segsAtoG;

component updown
      generic (test_mode : integer);
      port (clk, reset, enable, up_dn : in std_logic;
              AN3, AN2, AN1, AN0 : out std_logic;
          disp : out segsAtoG);
end component;

begin

uut : updown
 generic map(1)   -- sets test_mode to 1 to bypass clock divider
 port map(clk, reset, enable, up_dn, AN3, AN2, AN1, AN0, disp);

initial: process
begin
      reset <= '1';
      wait for 100 ns;
      reset <= '0';
      wait;            -- process hangs here forever
end process;

clkgen : process
begin
      clk <= '0';
      wait for 50 ns;
      clk <= '1';
      wait for 50 ns;
end process;

end behave;

configuration tb_cfg of tb is
for behave
end for;
end tb_cfg;
```