INTRODUCTION

This tutorial guide is an introduction to digital logic simulation and synthesis using the Mentor Graphics (Modelsim and Precision RTL) and Xilinx (ISE and Impact) tools. You should have working knowledge of the Linux operating system (using text editors, copying files, creating directories, printing, etc.). Knowledge of the VHDL language is not required to complete this tutorial. The VHDL code for every example has been included. The examples have been kept simple, the focus is on learning the tools rather than learning how to write VHDL code. There are many fine books dealing with VHDL; there are not so many books dealing with simulation and synthesis tools. This tutorial attempts to bridge the gap between the novice users' knowledge of such tools and the documentation available from the tool vendors. This tutorial is not meant as a definitive guide to the tools, rather it gently introduces the student to the many facets of the tools. It is hoped that after having completed the material contained in this guide that the on-line vendor documentation will not appear as foreboding and intimidating.

The tutorial is divided into five parts. Part I deals with VHDL simulation using the Modelsim simulator from Mentor Graphics Corporation. Part II focuses on logic synthesis; the Precision Synthesis software from Mentor Graphics being the tool of choice. Part III concerns itself with implementation using the Xilinx ISE software. In this section, the netlist file obtained as a result of the synthesis step (performed in Part II) is converted into physical hardware which (hopefully) functions correctly. Part IV gives details of the Xilinx FPGA demonstration board. This is the board which will be used to program and test the field-programamble gate array. Part V describes the use of the Mentor and Xilinx tools using the command line interface.

The examples in this tutorial were simulated, synthesized and downloaded to the FPGA demonstration board using Modelsim SE version 6.6d , Precision RTL Synthesis 2010a from Mentor Graphics, and Xilinx ISE 9.2i.

PART I : VHDL Simulation using MODELSIM

This section explains the use of the Modelsim tools to perform simulation of source code written in the VHDL langauge. Several examples will illustrate various aspects of the different tools available. Most of the tools are available in command-line version and also in graphical-user interface mode.

I. Setting up the user environment to run the Modelsim VHDL simulation tools.

Throughout this tutorial the Linux prompt is indicated by:

ted@brownsugar ~ 11:27am >

Your prompt may appear different depending upon the configuration of your account. Note also that the Linux hostname (brownsugar in the above example) changes from section to section in this tutorial as it was developed while running the various software from several different ECE workstations.

Prior to running the Modelsim tools, it is necessary to set up your Linux computer account. Perform the following from your Linux prompt:

Step 1:

ted@brownsugar ~ 11:27am > source /CMC/ENVIRONMENT/modelsim.env

Alternatively, one may copy the file /CMC/ENVIRONMENT/synopsys.env to one's home directory and source it from there (make sure you have the most recent version of the file):

ted@brownsugar ~ 11:27am > cd
ted@brownsugar ~ 11:27am > cp /CMC/ENVIRONMENT/modelsim.env **.**
ted@brownsugar ~ 11:27am > source modelsim.env

It is necessary to source the modelsim.env file every time you login in, or whenever you open a new terminal window.

Step 2:

We will first create a directory called `Modelsim`, and within this directory a subdirectory called `Code` will be created. The `Code` subdirectory will be used to contain the VHDL code to be simulated, and a directory called `work` (which will be used to hold intermediate files created by the simulation tools). The `work` directory will be created using a special Modelsim command (the **vlib** command) Figure 1 illustrates the directory hierarchy which will be created.

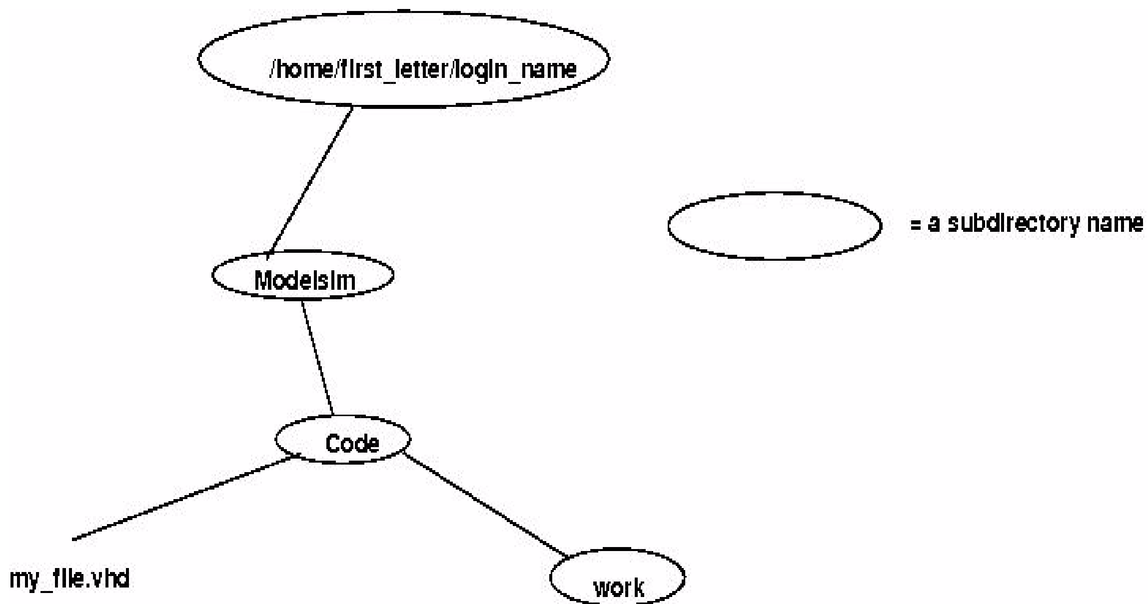


Figure 1: Directory hierarchy for VHDL Simulation using Modelsim

Issue the following commands from the UNIX prompt:

ted@brownsugar ~ 11:27am > cd
ted@brownsugar ~ 11:27am > mkdir Modelsim
ted@brownsugar ~ 11:27am > cd Modelsim
ted@brownsugar ~ 11:27am > mkdir Code
ted@brownsugar ~ 11:27am > cd Code

This sequence of commands will create the Modelsim directory and the Code subdirectory. We will now create the work directory used by the simulator:

ted@deadflowers Code 2:17pm >vlib work

This command will create a subdirectory called work and will create a file called `_info` containing some setup information used by the simulator. Do not delete the `_info` file as it is needed by the simulator. The above steps need only be performed one time.

This completes the setup for perfroming VHDL simulation using the Modelsim tools. In the next section we will present several examples on how to use Modelsim to perform VHDL simulation.

II. Performing VHDL simulation using Modelsim

This section will illustrate the use of the Modelsim tools used to perform VHDL simulation. The examples will illustrate various features of the tools.

Example 1: Simulating a 2-input AND gate (using the Graphical User Interface).

(1) Change into your `Modelsim/Code` directory and create a file called `and2.vhd` with the following contents:

```
entity and2_gate is
port( in_1, in_2: in bit;
      output    : out bit);
end;

architecture example of and2_gate is
begin
  output <= in_1 and in_2;
end;
```

You can use any Linux text editor (vi, emacs, nedit, gedit, etc) to create and save this file.

The next step is to “compile” the VHDL source file. This is a process similar to compiling source code written in a high-level programming language such a C++ or FORTRAN. During compilation of VHDL code, any syntactical errors will be reported.

The Modelsim tool used to compile VHDL source code is called **vcom** .

(2) Analyze the and2.vhd file using **vcom.**

ted@deadflowers Code 2:23pm >vcom and2.vhd
Model Technology ModelSim SE vcom 6.6d Compiler 2010.11 Nov 1 2010
-- Loading package standard
-- Compiling entity and2_gate
-- Compiling architecture example of and2_gate
ted@deadflowers Code 2:24pm >

A small message giving the version number of the tool will be displayed along with some information pertaining to the VHDL code being compiled and you will be returned to theLinux prompt if your code contains no syntax errors.

If there are syntactical errors in the source code, vcom will report the line number and attempt to describe the source of the error. For example, suppose that in the line port(in_1, in_2: in bit; the word bit was misspelled as bitt:

ted@deadflowers Code 2:25pm >vcom and2.vhd
Model Technology ModelSim SE vcom 6.6d Compiler 2010.11 Nov 1 2010
-- Loading package standard
-- Compiling entity and2_gate
** Error: and2.vhd(3): (vcom-1136) Unknown identifier "bitt".

```
** Error: and2.vhd(5): VHDL Compiler exiting
```

If you encounter syntax errors in your VHDL source code, correct them and recompile the code. The vcom command will compile the VHDL source code and create some intermediate files in the work directory:

```
ted@deadflowers Code 2:31pm >cd work
ted@deadflowers work 2:31pm >ls -al
total 20
drwx------ 4 ted ted 4096 Jul 20 14:30 .
drwx------ 3 ted ted 4096 Jul 20 14:29 ..
drwx------ 2 ted ted 4096 Jul 20 14:30 and2_gate
-rw------- 1 ted ted  361 Jul 20 14:30 _info
drwx------ 2 ted ted 4096 Jul 20 14:30 _temp
-rw------- 1 ted ted   26 Jul 20 14:30 _vmake
```

(3) The next step is to simulate the VHDL model. This is done with the **vsim** command From the Linux enter:

```
ted@deadflowers Code 2:33pm >vsim and2_gate &
```

The name following the vsim command refers to the entity name that has been compiled into the work library.

You will see lines similar to the following:

```
[1] 28373
ted@deadflowers Code 2:41pm >Reading /nfs/sw_cmc/linux-32/tools/mentor.2011/
modelsim_6.6d/modeltech/tcl/vsim/pref.tcl
```

Three windows will open as shown in Figure 2. These are the Objects window, the main Modelsim simulation window and and Wave window. The Objects windows shows the names and values of data objects in the current main simulation window. In this example, the Objects windows shows the two input ports (in_1 and in_2) and the output port and indicates their mode (in or out) and that they are of type signal. Simulation commands are entered in the Transcript portion (bottom portion) of the Simulation window and the results are displayed in thwe Waveform window.

Before entering and simulation commands, it is necessary to add any signals you wish to have displayed in the Waveform window. This is done by moving the mouse cursor over the Add choice of the top menu bar of the Objects window and selecting:

Add ----> To Wave ----> Signals in Region.

You will now see listed in the Wave window the three ports of the design. Refer to Figure 3.

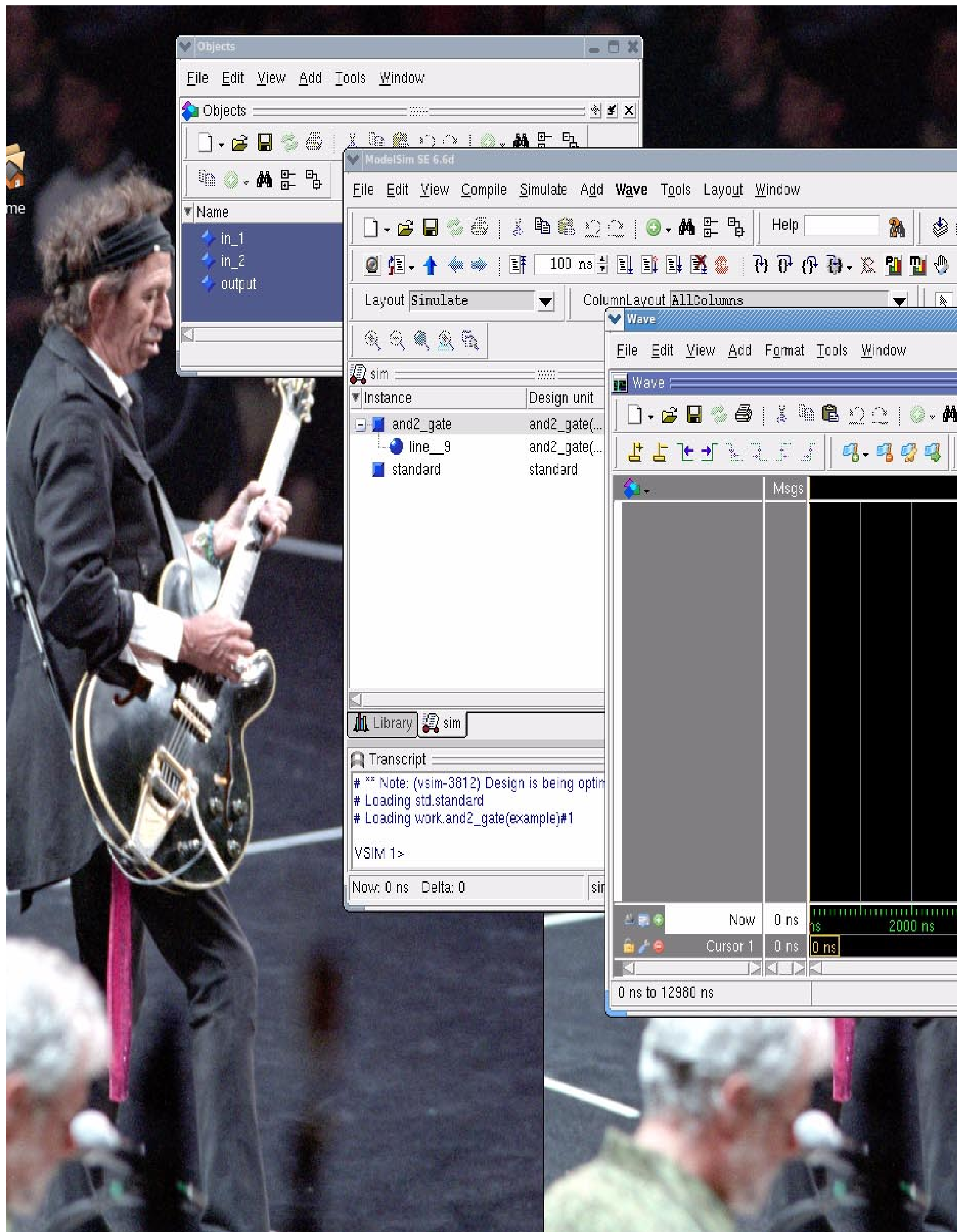


Figure 2: Modelsim simulation windows.

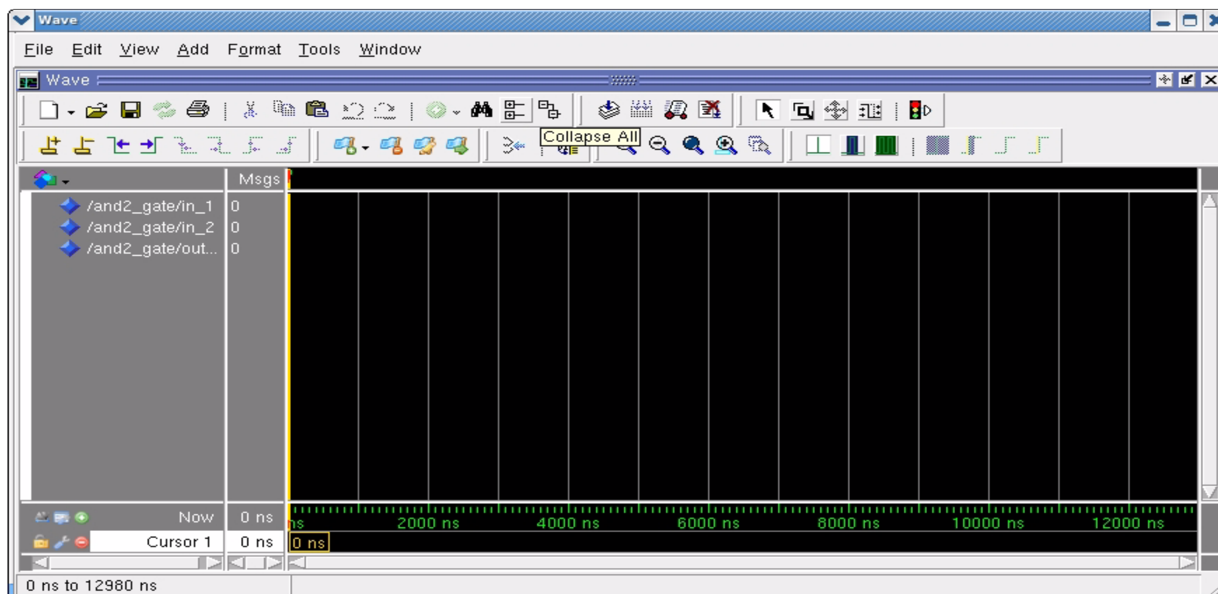


Figure 3: Wave window after signals have been added.

(5) One can manually assign values to signals using the **force** command. This command is issued from the Transcript portion of the Simulation window. Enter the following commands from the bottom portion of the Simulation window as shown in Figure 4:

force in_1 0
force in_2 0
run

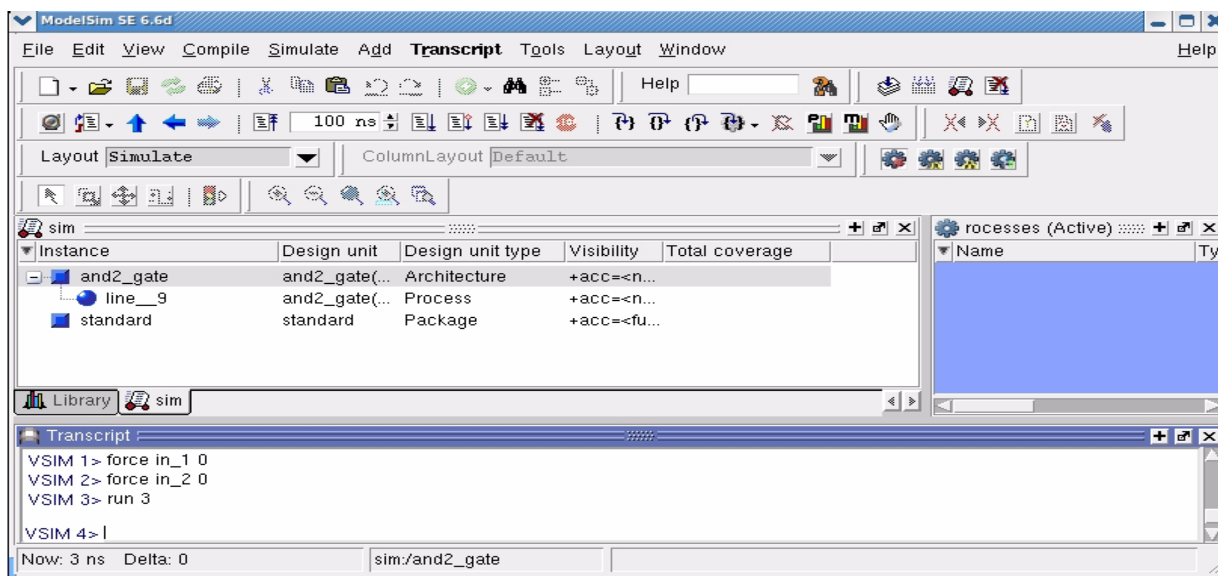


Figure 4: Issueing a force command from the Simulation window.

The force command assigns a signal the specified value. The signal keeps this value until a new force command sets it to another value. The run command advances simulation time by the specified number of timesteps.

Examine the waveform in the Wave window. One may zoom to a full view format by selecting:

View ---> Zoom ---> Zoom Full

from the Wave window.

Use the force command to set the inputs to their different possible values:

force in_1 0
force in_2 1
run 2

force in_1 1
force in_2 0
run 2

force in_1 1
force in_1 1
run 2

Figure 5 shows the results of the simulation.

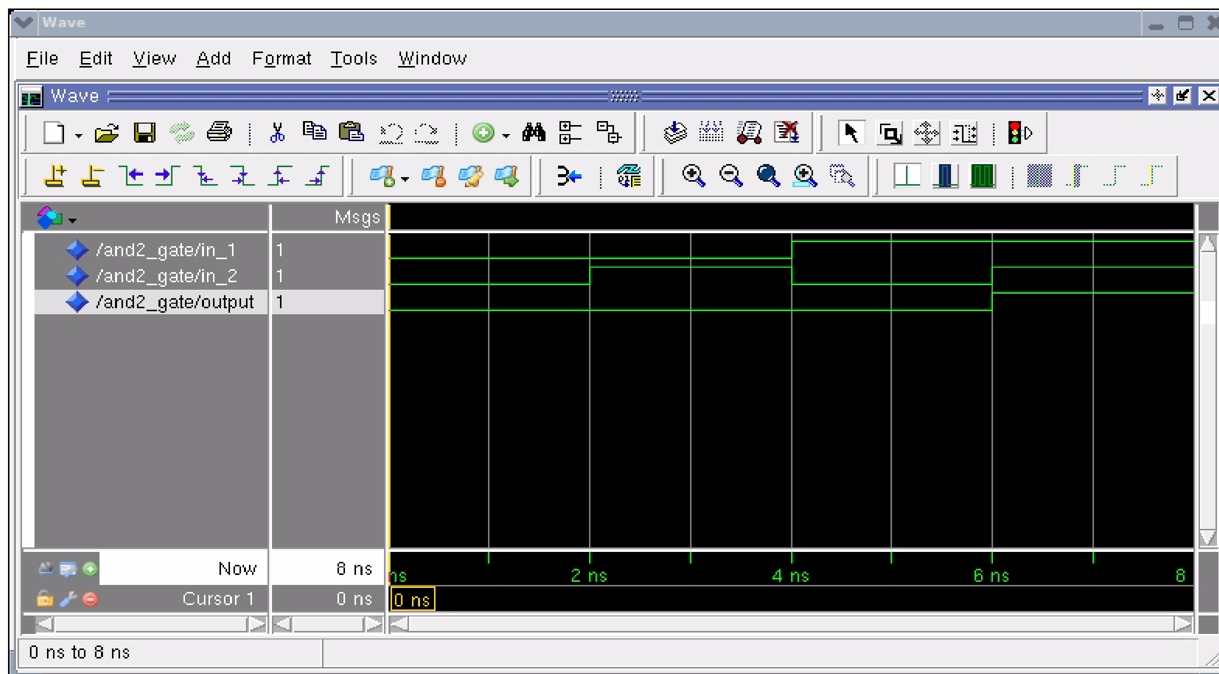


Figure 5: The Wave window containing the simulation results.

(6) To obtain a printout of the Wave window, select File -> Print Postscript. In the Write Postscript window, specify whether to print to a printer or to print to a file. One can also specify the time range to print. Select OK to generate the file. Refer to Figure 6 for the details of setting the various print options.

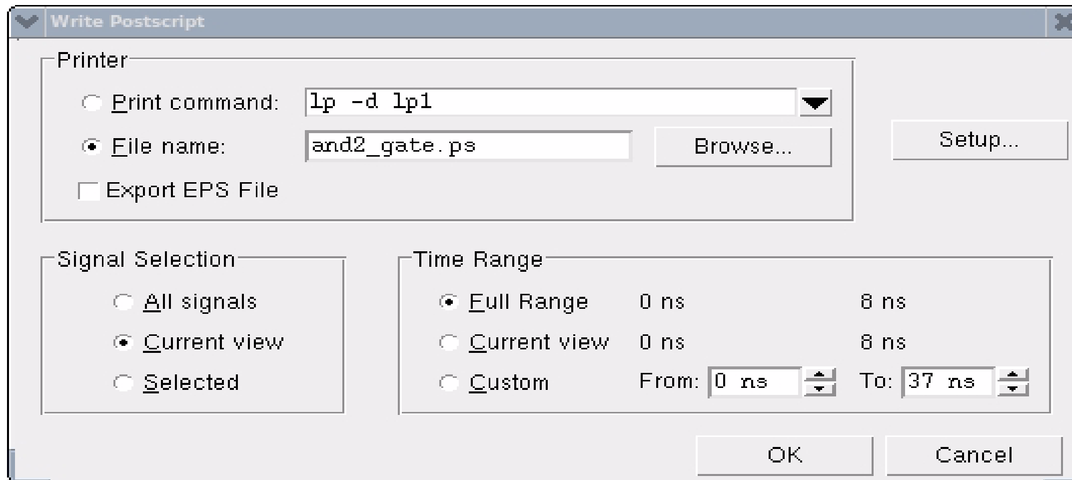


Figure 6: Setting the Print options.

To print the file to a printer, use the Linux command:

lpr -Pprinter_name filename

where printer_name is the name of the printer you wish to print to.

To quit the simulator, select File -> Quit from the Simulation window.

This concludes this introductory example.

<u>Example 2:</u> Simulating a two-input and gate (using a DO file)

Entering commands through the simulator prompt can be tedious, especially if it necessary to enter the commands a number of times. It is possible to store theses commands in a file and have the simulator read and execute these commands from the file. This example makes use of the same and2_gate.vhd file, but uses a DO file to store the commands in.

Create a subdirectory called DO within your Modelsim directory and within this subdirectory create a text file called and2.do containing the following:

```
add wave in_1
add wave in_2
add wave output
force in_1 0
```

```
force in_2 0
run 2
force in_1 0
force in_2 1
run 2
force in_1 1
force in_2 0
run 2
force in_1 1
force in_2 0
run 2
```

Once can invoke the simulator specifying that it should read and execute the contents of a specific DO file using the following syntax:

ted@deadflowers Code 5:59pm >vsim -do ../DO/and2.do and2_gate &

The -do command line option is used to specify the path to the DO file to be read in. The .. is Linux shorthand notation for the parent of the present working directory. Alternatively, one may save the DO files in any directory and simply specify the complete pathname to the location of the specific DO file. The three windows will appear and the simulation will run to completion. Zoom to a Full view in the Wave window and you will oberve the same simulation results as obained earlier in Example 1.

An alternative method of reading in a DO file is the first load the design into the simulator:

ted@deadflowers Code 5:59pm >vsim and2_gate &

and then from the Simulation Transcript window, enter the following command:

do ../DO/and2.do

<u>Example 3:</u> Simulating a design consisting of multiple VHDL source code files.

This example illustrates the use of DO files and also illustrates the use of a multi-level hierarchical design style. Three separate VHDL source files will be analyzed, and the top-level entity will make references to the the lower-level files. The two bottom-level files specify the entity-architecture pairs for an AND gate and a OR gate respectively.. The top-level entity consists of a combinational logic circuit consisiting of two AND gates and a single OR gate.

(1) Create a file called tedand.vhd (save it in your Code directory) with the following contents:

```
entity ted_and is
port(A,B : in BIT ; OUTPUT : out BIT);
end ted_and;
```

```
architecture ted_arch of ted_and is
begin

OUTPUT <= A and B after 5 ns;

end ted_arch;
```

(2) Create a file called tedor.vhd with the following VHDL statements in it:

```
entity ted_or is
port(A,B : in BIT ; OUTPUT : out BIT);
end ted_or;

architecture ted_arch of ted_or is
begin

OUTPUT <= A or  B;

end ted_arch;
```

(3) Create a file called tedcircuit.vhd with the following contents (this will be our top-level entity):

```
entity tedcircuit is
port(A,B,C,D : in BIT; E : out BIT);
end tedcircuit;

architecture ted_arch of tedcircuit is

-- declare the components found in our entity

component ted_and
port(A, B : in BIT; OUTPUT : out BIT);
end component;

component ted_or
port(A,B : in BIT; OUTPUT : out bit);
end component;

-- declare signals used to interconnect components

signal s1, s2 : BIT;

-- declare configuration specification

for U1, U3 : ted_and use entity WORK.ted_and(ted_arch);
for U2: ted_or  use entity WORK.ted_or(ted_arch);
```

```
begin

U1 : ted_and port map(A => A , B => B , OUTPUT => s1 );
U2 : ted_or  port map(A => C,  B => D,  OUTPUT => s2 );
U3 : ted_and port map(A => s1, B => s2, OUTPUT => E);

end ted_arch;
```

(4) Note how the entity tedcircuit (whose architecture is specified in the file tedcircuit.vhd) makes references to entities whose architecture is specified in a separate file. Specifically, the two components ted_and and ted_or are specified in two separate files. The rules specifying the order of compilation of VHDL units require that the two files tedand.vhd and tedor.vhd be compiled prior to the compilation of the file tedcircuit.vhd. We would compile these three files in the following order:

```
ted@deadflowers Code 6:18pm >vcom tedand.vhd
Model Technology ModelSim SE vcom 6.6d Compiler 2010.11 Nov  1 2010
-- Loading package standard
-- Compiling entity ted_and
-- Compiling architecture ted_arch of ted_and
ted@deadflowers Code 6:18pm >vcom tedor.vhd
Model Technology ModelSim SE vcom 6.6d Compiler 2010.11 Nov  1 2010
-- Loading package standard
-- Compiling entity ted_or
-- Compiling architecture ted_arch of ted_or
ted@deadflowers Code 6:18pm >vcom tedcircuit.vhd
Model Technology ModelSim SE vcom 6.6d Compiler 2010.11 Nov  1 2010
-- Loading package standard
-- Compiling entity tedcircuit
-- Compiling architecture ted_arch of tedcircuit
-- Loading entity ted_and
-- Loading entity ted_or
ted@deadflowers Code 6:18pm >
```

(5) Create a DO file (save it in your DO directory) called tedcircuit.do which contains the following:

```
# add all the signals to the wave window
add wave *
# setup some input values and run the simulator

force a 0
force b 0
```

```
force c 0
force d 0
run 2

force a 0
force b 1
force c 0
force d 1
run 2

force a 1
force b 1
force c 0
force d 1
run 2

force a 1
force b 1
force c 1
force d 1
run 2
```

Note how comments within a DO file are specified using the # character as the first character of the line. Instead of explicitly adding all the signals, one can use the wildcard character * to add all the signals contained in a given entity to the Wave window.

(6) Load the compiled design into the simulator together with the specified DO file:

ted@deadflowers Code 6:25pm >vsim -do ../DO/tedcircuit.do tedcircuit &

Figure 7 shows the simulation results for this example.

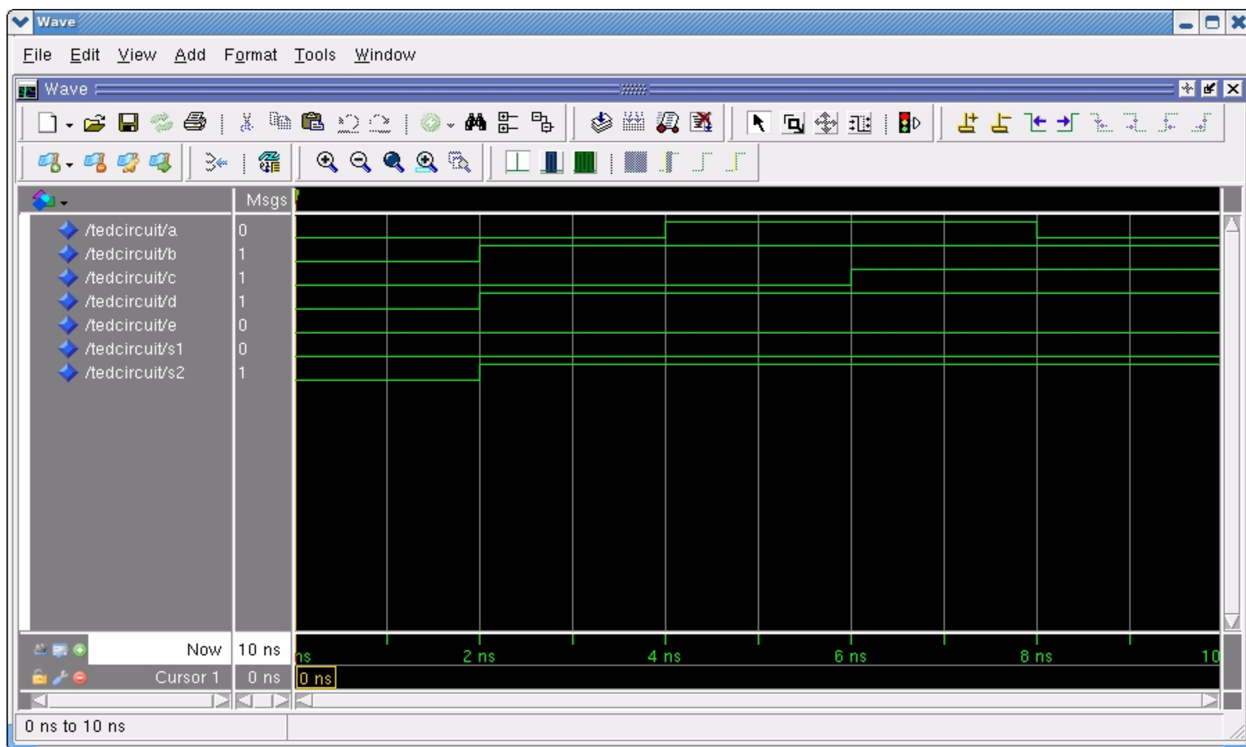


Figure 7: Simulation results for Example 3.

<u>Example 4:</u> Specifying repeating signals.

When simulating VHDL designs, it is often neccessary to specify a repeating pattern for a certain signal such as a clock input to a synchronous system. There are several methods of doing so through a DO file. This example will illustrate three methods ranging from a brute-force approach to a more consice and refined manner. The three methos make use of the following VHDL code which describes a simple 3-bit counter with an asynchronous reset (active-low), and a count enable signal.

1. Create the following VHDL code in your Code directory with the filename count3.vhd:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity count3 is
  port( clk, resetn, count_en : in std_logic;
        sum                   : out std_logic_vector(2 downto 0);
        cout                  : out std_logic);
end count3;
```

```
architecture rtl of count3 is
signal count : std_logic_vector(2 downto 0);
begin

  process(clk, resetn)
  begin
    if resetn = '0' then
       count <= (others => '0');
    elsif clk'event and clk = '1' then
       if count_en = '1' then
          count <= count + 1;
       end if;
    end if;
  end process;

  sum <=  not count; -- invert the outputs for the demo board
                     -- since its LEDs are active low


  cout <= '0' when count = 7 and count_en = '1' else '1';


end rtl;
```

2a: The “brute-force” DO file.

Create the following DO file in your DO directory called count3.do:

This is a comment line in a .do file

```
# add all signals to the Waveform window
add wave *

# apply a reset to the counter
force resetn 0
force clk 0
force count_en 1
run 2

#  unassert the reset signal and clock
#  for several cycles
force resetn 1
run 2

force clk 1
```

```
run 2
force clk 0
run 2

force clk 1
run 2
force clk 0
run 2

force clk 1
run 2
force clk 0
run 2

force clk 1
run 2
force clk 0
run 2

force clk 1
run 2
force clk 0
run 2

force clk 1
run 2
force clk 0
run 2

force clk 1
run 2
force clk 0
run 2

force clk 1
run 2
force clk 0
run 2

force clk 1
run 2
force clk 0
run 2
```

This simplistic DO file simply asserts a reset pulse, then applies 9 clock pulses in repetition. Clearly, one would not want to adopt such a manner if it were necessary to simulate a design over

hundreds of clock cycles... there are far easier ways of doing this as shown in 2(b).

2(b) A DO file which makes use of another DO file.

Create the following two DO files called clock.do and count3b.do respectively:

clock.do:

```
# toggle to clock between 1 and 0

force clk 1
run 2
force clk 0
run 2
```

count3b.do:

```
# This .do file reads in another .do file which
# toggles the clock signal for 9 cycles

# add all signals to the Waveform window
add wave *

# apply a reset to the counter
force resetn 0
force clk 0
force count_en 1
run 2

#  unassert the reset signal and clock
#  for several cycles
force resetn 1
run 2

do clock.do
do clock.do
do clock.do
do clock.do
do clock.do
do clock.do
do clock.do
do clock.do
do clock.do
```

Simulate the design:

ted@deadflowers Code 7:20pm >vsim -do ../DO/count3b.do count3 &

The first part of this DO file is similar to the brute-force one, it differs in that it reads in the clock.do file 9 times instead of repeating the

```
force clk 1
run 2
force clk 0
run 2
```

statements explicitly 9 times. This is a slight improvement over the brute-force method, but still awkward to use if it is necessary to simulate a design over many clock cycles.

2(c:) Using a force command with a repeat.

Create the following DO file called count3c.do:

```
# This is a comment line in a .do file
# add all signals to the Waveform window
add wave *

# apply a reset to the counter
force resetn 0
force clk 0
force count_en 1
run 2

#  unassert the reset signal and clock
#  for several cycles
force resetn 1
run 2

force clk 1  2 -r 4
force clk 0  4 -r 4

# run for 9 clock periods
# 9 clock periods x 4 timesteps per period
# = 36 timesteps

run 36
```

The two force commands make use of the -r option which is used to repeat the action.
The syntax of this force command is:

force signal_name   signal_value   start_time -r repeat_time

Thus, the first force command will set the clk signal to a value of 1 at a time equal to 2 time units after the current simulation time and this will be repeated at a time commencing at 4 time units after the current simulation time. In a similar manner, the second force command will set the clk signal to a value of 0 at a time equal to 4 units after the current simulation time and this will be repeated starting at 4 time units after the current simulation time. Note that the two force commands may be combined into one in the following manner:

```
force clk 1  2, 0  4 -r 4
```

This command forces the signal called clk to a value of 1 at a time value of 2 units after the current simulation time, and then it forces the signal to value 0 at a time equal to 4 units after the current simulation time and this cycle is repeated at time 4 units after the current simulation time.[1]

This method is the most practical for creating a periodic clock signal over many cycles.

Figure 8 shows the simulation results obtained from using the count3c.do file.

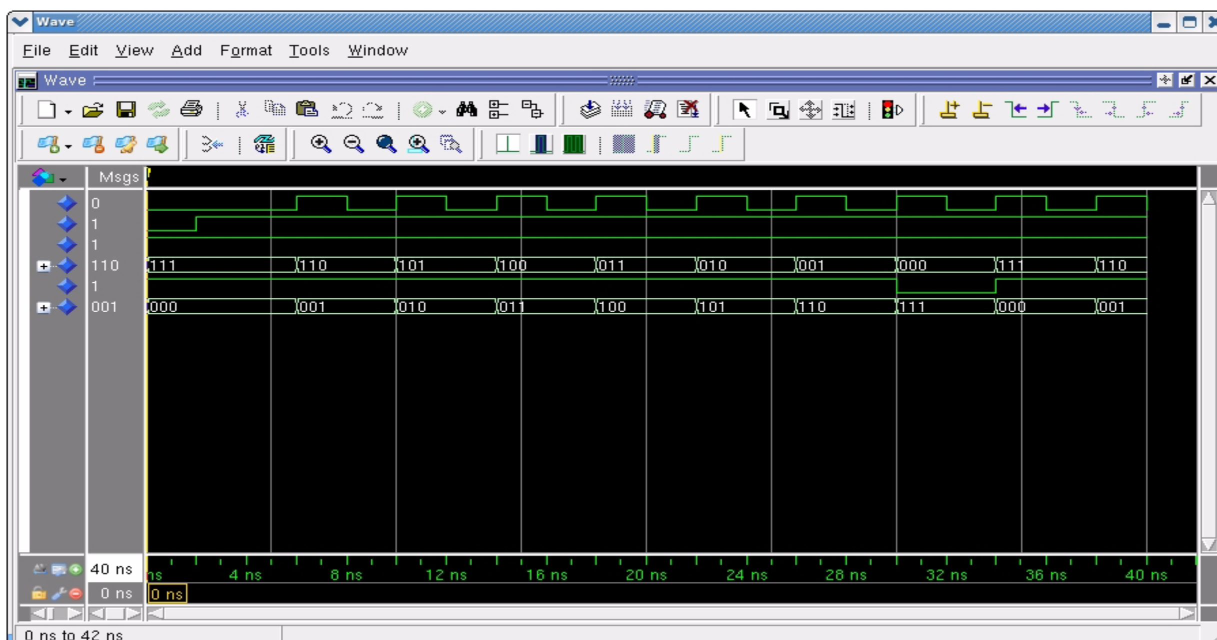


Figure 8: Simulation results obtained with the repeated force command.

PART II : Logic Synthesis with Precision® RTL

In this section we will use the Mentor Graphics Precision RTL tool to perform logic synthesis. In synthesis, VHDL code will be translated into an EDIF (Electronic Design Interface Format) netlist file. This netlist file can then be used as input to third-party implementation tools. In this tutorial we will be using the Xilinx ISE tool suite which will perform the translation from EDIF netlist f into an implemented design downloaded to a FPGA development board.

I. Performing Logic Synthesis

This section will explain the use of the Precision® RTL Synthesis tool from Mentor Graphics Corporation. To quote the User's Manual, "Precision™ RTL is a comprehensive tool suite, providing design capture in the form of VHDL and Verilog entry, advanced register-transfer-logic logic synthesis, constraint based optimization, and schematic viewing." [2]

You will have to create a subdirectory called FPGA_ADV (from within your Modelsim directory) to hold the files created by the Precision RTL synthesis tool. This directory may be created in the following manner:

ted@brownsugar Code 1:23pm > cd   (this will return you to your home directory)
ted@brownsugar ~ 1:23pm > cd Modelsim
ted@brownsugar Modelsim 1:23pm > mkdir FPGA_ADV

Example 1: Synthesizing a structural VHDL design.

This example consists of a full adder circuit constructed from two half adders and an OR gate. The port map statement is used to instantiate two instances of a half-adder component. Note that the top level output ports (sum_out_neg, carry_out_neg) have been negated since the FPGA board LED's are active LOW (this means the LED is illuminated when it is driven by a logic 0).

(1) Create the following files in your Code directory:

(i) a file called half_adder_regular_outputs.vhd with the following contents:

```
library ieee;
use ieee.std_logic_1164.all;

entity half_adder is
    port ( in1, in2 :   in std_logic;
           carry, sum : out std_logic);
end half_adder;

architecture true_outputs of half_adder is
begin
  carry <=  (in1 and in2);
```

```
  sum   <=  (in1 xor in2);
end true_outputs;
```

(ii) a file called full_adder_negated_outputs.vhd with the following contents:

```
library ieee;
use ieee.std_logic_1164.all;

entity full_adder_negated is
  port(carry_in, input1, input2 : in std_logic;
       sum_out_neg, carry_out_neg : out std_logic);
end full_adder_negated;

architecture structural of full_adder_negated is

-- declare a half-adder component

component half_adder
  port ( in1, in2 :   in std_logic;
         carry, sum : out std_logic);
end component;

-- declare internal signals used to "hook up" components

signal carry1, carry2     : std_logic;
signal sum_int            : std_logic;
signal sum_out, carry_out : std_logic;

-- declare configuration specification

for ha1, ha2 : half_adder use entity WORK.half_adder(true_outputs);

begin

-- component instantiation

ha1: half_adder port map(in1 => input1, in2 => input2,
                         carry => carry1, sum => sum_int);

ha2: half_adder port map(in1 => sum_int, in2 => carry_in,
                         carry => carry2, sum => sum_out);

carry_out <=  carry1 or carry2;

-- negate the internal sum and carry to the external port signals
-- since the XUP Virtex2 Pro demo board has active LOW LED outputs
-- DIP switch in UP position will produce a logic-'0' value.

carry_out_neg <=  not carry_out;
sum_out_neg   <=  not sum_out;

end structural;
```

(2) The next step is to setup your Linux environment to run the Precision RTL synthesis tool. This is done by sourcing the setup file /CMC/ENVIRONMENT/fpga_advantage.env as shown below: Change into you FPGA_ADV directory and then source the file:

ted@deadflowers FPGA_ADV 12:43pm >source /CMC/ENVIRONMENT/fpga_advantage.env

After this command is issued, you will be returned back to your Linux prompt. As a double check to ensure that your environment is configured properly, issue the which precision command:

ted@deadflowers FPGA_ADV 12:43pm >which precision
/encs/pkg/Precision-2010aU1/root/Mgc_home/bin/precision

Note that this tutorial has been written using version 7.1 of Precision RTL. This may change as newer versions are installed. The actual results returned from the 'which' command may vary from the ones given above.

(3) Invoke the Precision RTL tool:

ted@brownsugar FPGA_ADV 1:24pm > precision

The following will be displayed and two windows shown in Figures 9 and 10 will appear:

ted@deadflowers FPGA_ADV 2:22pm >precision: Setting MGC_HOME to /encs/pkg/Precision-2010aU1/root/Mgc_home ...
precision: Executing on platform: Scientific Linux SL release 5.5 (Boron) 2.6.18-238.12.1.el5 i686

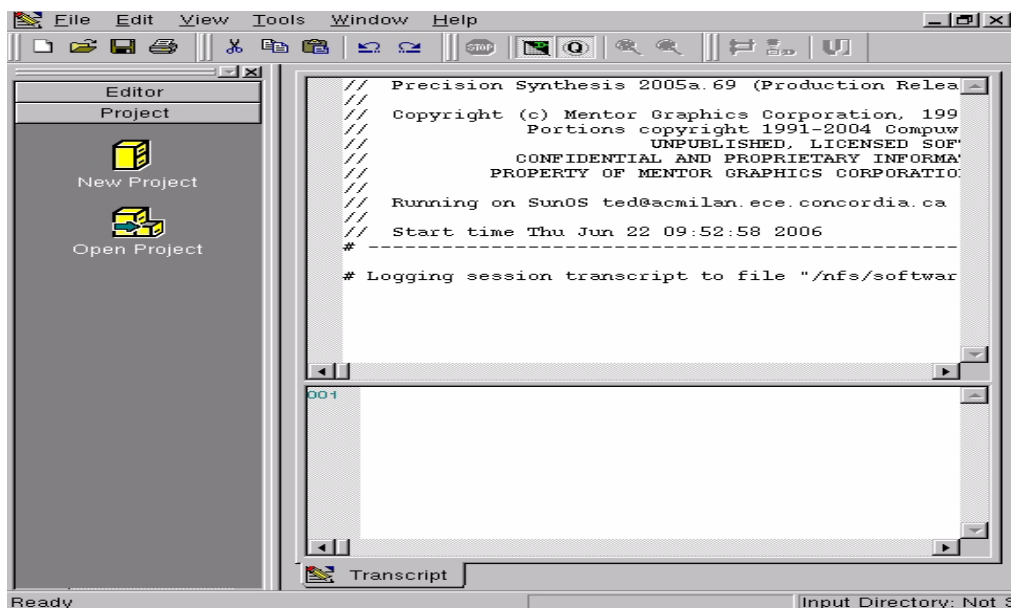


Figure 9: Main window for the Precision RTL synthesis tool.

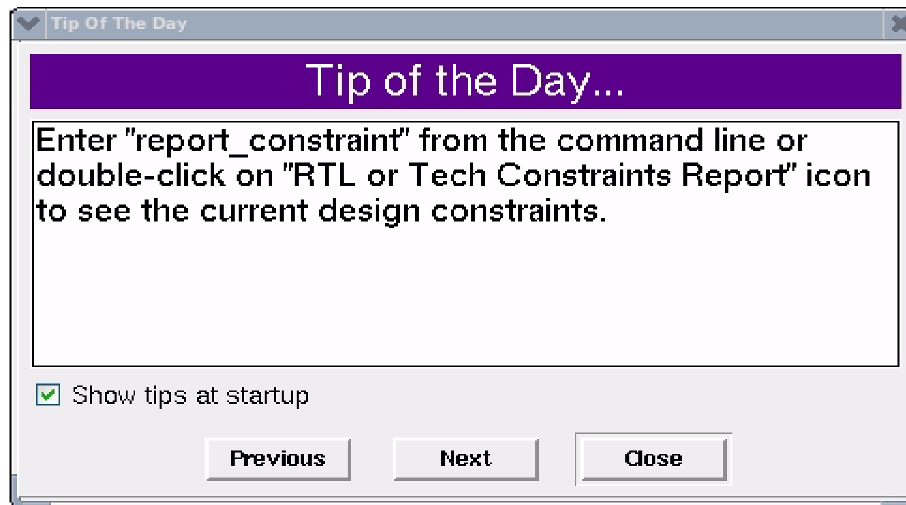


Figure 10: Tip of the Day window.

Click on the **Close** button in the Tip of the Day window.

(4) Select the New Project icon and fill out the New Project form specifying the following:

Project Name: Full_Adder_Test
Project Folder: /nfs/home/first_letter_of_first_name/your_login_name/Modelsim/FPGA_ADV

The information in the **Create Impl:** field will change to what you have entered as the Project Name (Full_Adder_Test_impl). Refer to Figure 11 for the details. Click on the **OK** button once the form has been filled.

Project Name : Full_Adder_Test
Project Folder /nfs/home/t/ted/SYNOPSYS_2000/FPGA_A ...
Create Impl: Full_Adder_Test_i
OK Cancel

Figure 11: New Project form.

(5) The next step is to specify the VHDL files which to be synthesized. This is performed by

selecting the **Add Input Files** icon which appears in the left hand pane of the main Precision window. The specified files will be read into memory and used to build a database used by the synthesis tool. Precision will analyze all of the files together, consequently the order in which your VHDL files are specified is immaterial. Futhermore, the top-level entity will be automatically detected.

Selecting the **Add Input Files** will result in the **Open** form appearing. Specify the full_adder_negated_outputs.vhd file in the **File** field by using the Up arrow yellow folder icon to navigate in your Code directory and select the specified file. Click **OK**. Refer to Figure 12.

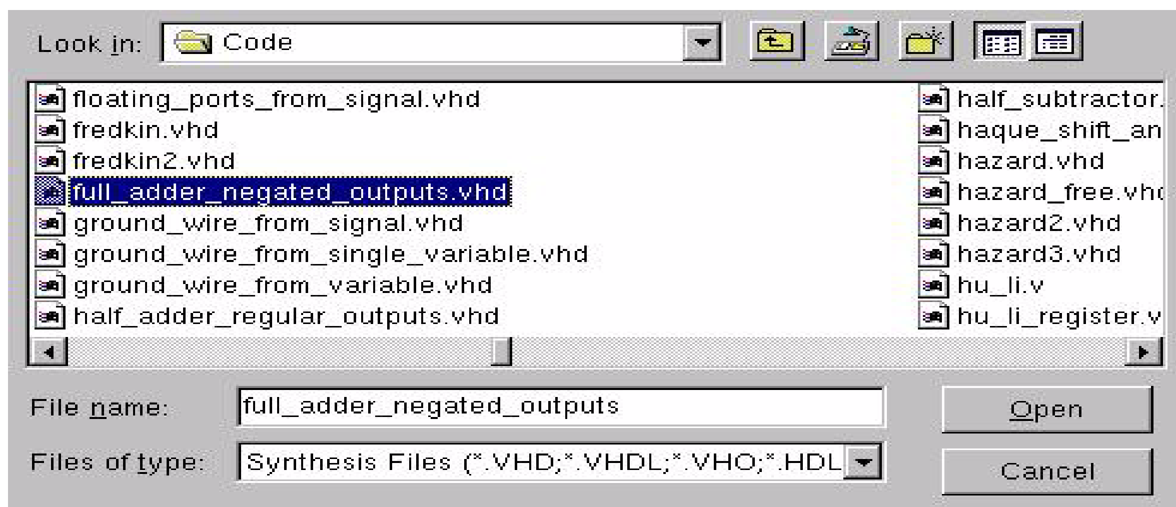


Figure 12: Open form.

(6) Repeat the above procedure to add the half_adder_regular_outputs.vhd file. You will now see that the files you specified are listed in the Project files pane of the main window as shown in Figure 13.

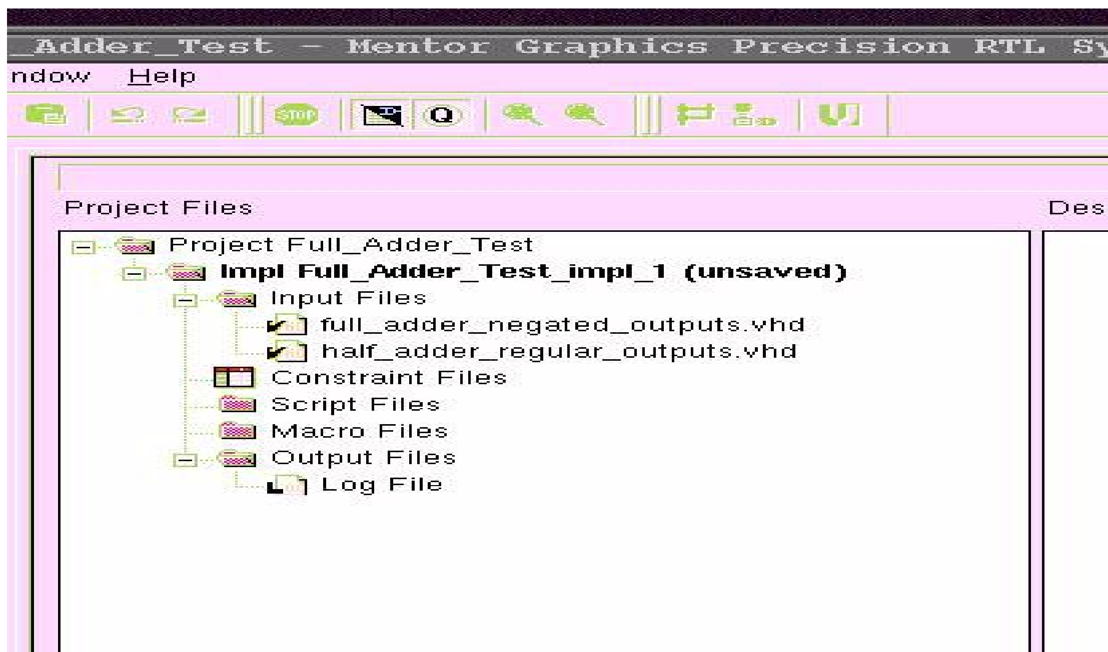


Figure 13: Project files pane of the main Precision window indicating added files.

(7) The next step is to specifiy the FPGA device you wish to use. This is performed by selecting the **Setup Design** icon. Clicking this icon will open the **Project Settings** form. In this form select Xilinx as the technology as shown in Figure 14.

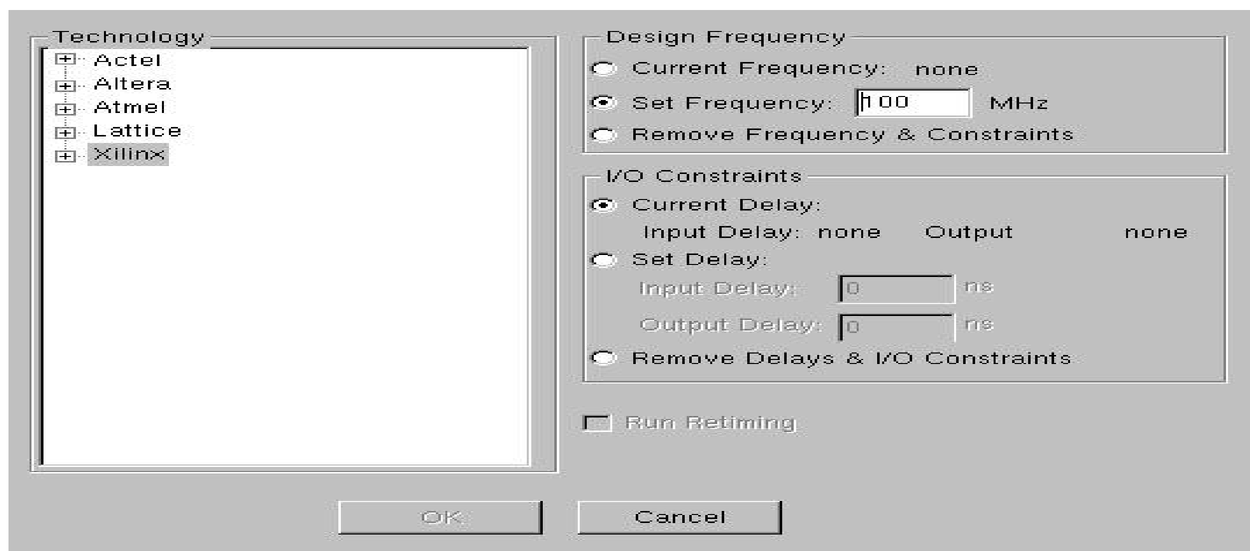


Figure 14: Project Settings Form.

Selecting the + symbol to the left of the word **Xilinx** will list the available Xilinx devices. Scroll through this list and select **VIRTEX-II Pro** as the family, **XC2VP30ff896** as the Device and **-7** as the speed grade. Click **OK** after you have specfied the values. This is the FPGA used in the development board the lab is equipped with. Refer to Figure 15 for the details of this form.

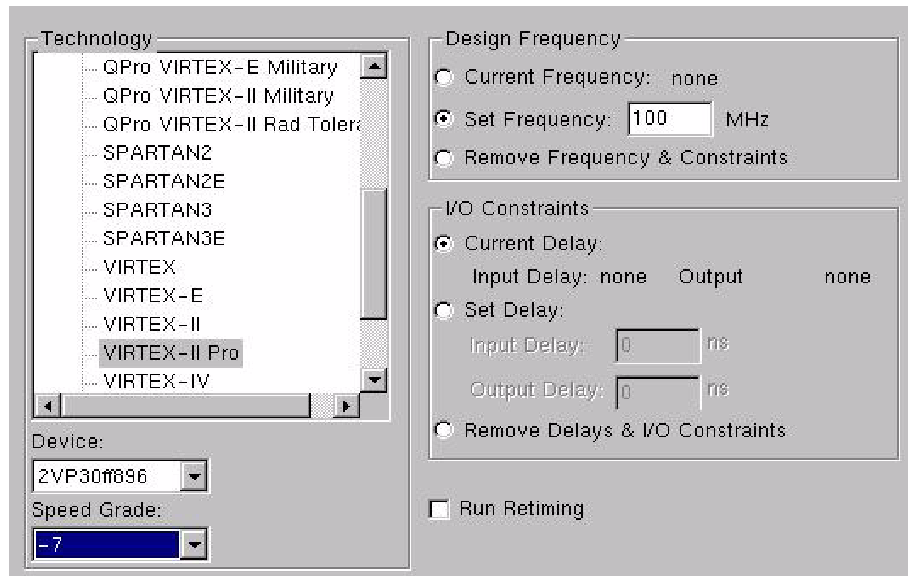


Figure 15: Specifying the Family, Device, and Speed grade.

(8) You are now ready to compile the .vhd files. You will note that there are now two new icons in the main window:

Select the **Compile** icon. As the Compile command executes, messages will be displaye din the middle pane of the window. Observe these messages for any warnings or error messages. These messages are also written to a log file with filename precision.log within the FPGA_ADV directory. The following is an excerpt of the contents of this log file:

```
# COMMAND: compile
```

```
#
# Info: Reading file: '/nfs/software/cmc/tools/MentorB.4/fa_71/Precision/
Mgc_home/pkgs/psr/techlib
s/xcv2p.syn'.
# Info: vhdlorder, Release 2005a.11
# Info: Files sorted successfully.
# Info: hdl-analyze, Release RTLC-Precision 2005a.11
# Info: 2502:  Analyzing input file "/nfs/home/t/ted/SYNOPSYS_2000/FPGA_ADV/
../Code/half_adder_reg
ular_outputs.vhd" ...
# Info: 2502:  Analyzing input file "/nfs/home/t/ted/SYNOPSYS_2000/FPGA_ADV/
../Code/full_adder_neg
ated_outputs.vhd" ...
# Info: Top of Design has been set to: full_adder_negated.
# Info: Current working directory: '/nfs/home/t/ted/SYNOPSYS_2000/FPGA_ADV/
Full_Adder_Test_temp_1/
'.
# Info: RTLC-Driver, Release RTLC-Precision 2005a.11
# Info: Last compiled on Jul  5 2005 15:52:47
# Info: 4512: Initializing...
# Info: 4504: Partitioning design ....
# Info: RTLCompiler, Release RTLC-Precision 2005a.11.2
# Info: Last compiled on Jul  7 2005 22:09:45
# Info: 4512: Initializing...
# Info: 4522: Root Module work.full_adder_negated(structural): Pre-process-
ing...
# Info: 4506: Module work.half_adder(true_outputs): Pre-processing...
# Info: 4508: Module work.half_adder(true_outputs): Compiling...
# Info: 4523: Root Module work.full_adder_negated(structural): Compiling...
# Info: 4842: Compilation successfully completed.
# Info: 4835: Total CPU time taken for compilation: 0.0 secs.
# Info: 4856: Total lines of RTL compiled: 59.
# Info: 4513: Overall running time 7.0 secs.
# Info: Current working directory: '/nfs/home/t/ted/SYNOPSYS_2000/FPGA_ADV/
Full_Adder_Test_temp_1/
'.
# Info: Finished compiling design.
compile
```

(9) The next step is to synthesize your design. Select the **Synthesize** icon in the left hand pane. Synthesis messages will appear in the middle pane. You may read them from the precision.log file if any errors occur. The following are the messages relevant to the Synthesize command from the log file:

```
# COMMAND: synthesize
#
# Info: Current working directory: '/nfs/home/t/ted/SYNOPSYS_2000/FPGA_ADV/
Full_Adder_Test_temp_1
'.
# Info: 2 Instances are flattened in hierarchical block
.work.full_adder_negated.structural.
# -- Optimizing design .work.full_adder_negated.structural
# Info: Starting a constant propagation on the mapped netlist.
```

```
# Info: Writing file: '/nfs/home/t/ted/SYNOPSYS_2000/FPGA_ADV/
Full_Adder_Test_temp_1/full_adder_negated.edf'.
# Info: Writing file: '/nfs/home/t/ted/SYNOPSYS_2000/FPGA_ADV/
Full_Adder_Test_temp_1/full_adder_ngated.ucf'.
# Info: Finished synthesizing design.
# /nfs/home/t/ted/SYNOPSYS_2000/FPGA_ADV/Full_Adder_Test_temp_1/
precision_tech.sdc
synthesize
ted@brownsugar FPGA_ADV 12:46pm >
```

(10) The last step is to **Save** the project. From the main Precision window select:

File : **Save Project**

The files which have been written into the Full_Adder_Test_temp_1 directory will be copied into the Full_Adder_Test_impl_1 directory. The Full_Adder_Test_temp_1 will still exist until you exit from Precision by selecting:

File : Exit (answer Yes when prompted “Are you sure you want to exit?”).

Once you have exited from Precision, the Full_Adder_Test_temp_1 will be deleted.

The end result of the Synthesize command is the generation of an EDIF netlist. This is highlighted in the above messages in boldface font. In this example, the name of the EDIF file is full_adder_negated.edf. This is an ASCII text file. You should examine the contents of this file so that you become familiar with its contents (look in the Full_Adder_Test_impl_1 directory)

(11) Upon completion of the Synthesize command, the middle pane of the main window will appear as shown in Figure 16.

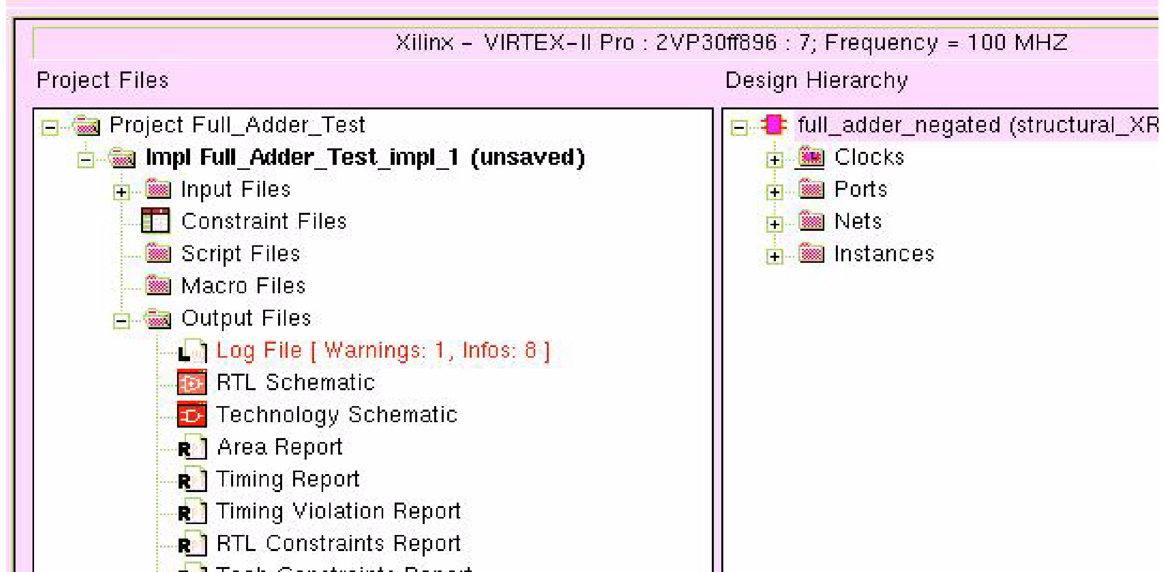


Figure 16: Main window after the Synthesize command.

To view the RTL schematic diagram of your synthesized hardware double click the **RTL Schematic** icon in the left hand pane of the middle pane. The schematic diagram will appear in the right hand pane of the middle pane as shown in Figure 17.

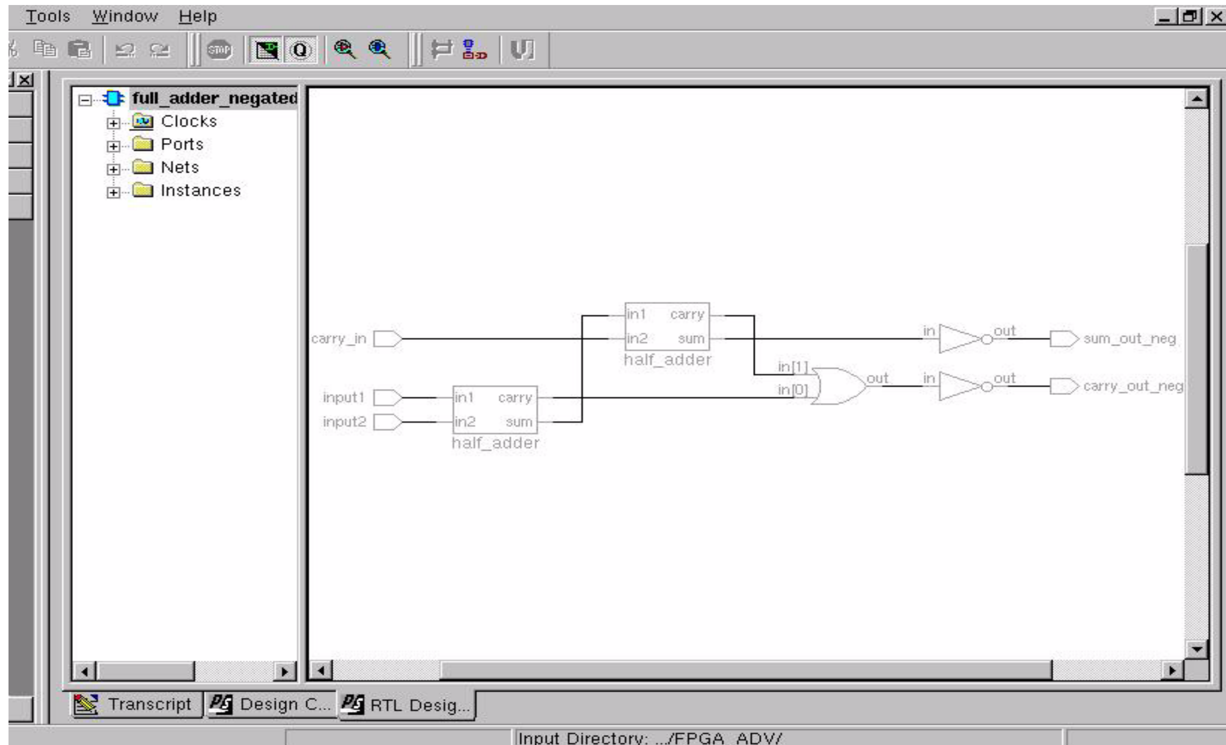


Figure 17: RTL Schematic of synthesized hardware.

The RTL Schematic window is very useful for analyzing and debugging a design if there are errors. For example, one may locate any net (a net in logic synthesis terminology is basically a wire connecting two terminals) by selecting the + symbol to the eft of the word Nets in the left hand portion of the pane. All the nets within the design will be listed. You may choose one of these (carry1 for example). The wire corresponding to this net will be highlighted in red in the schematic area of the window as inidicated in Figure 18.

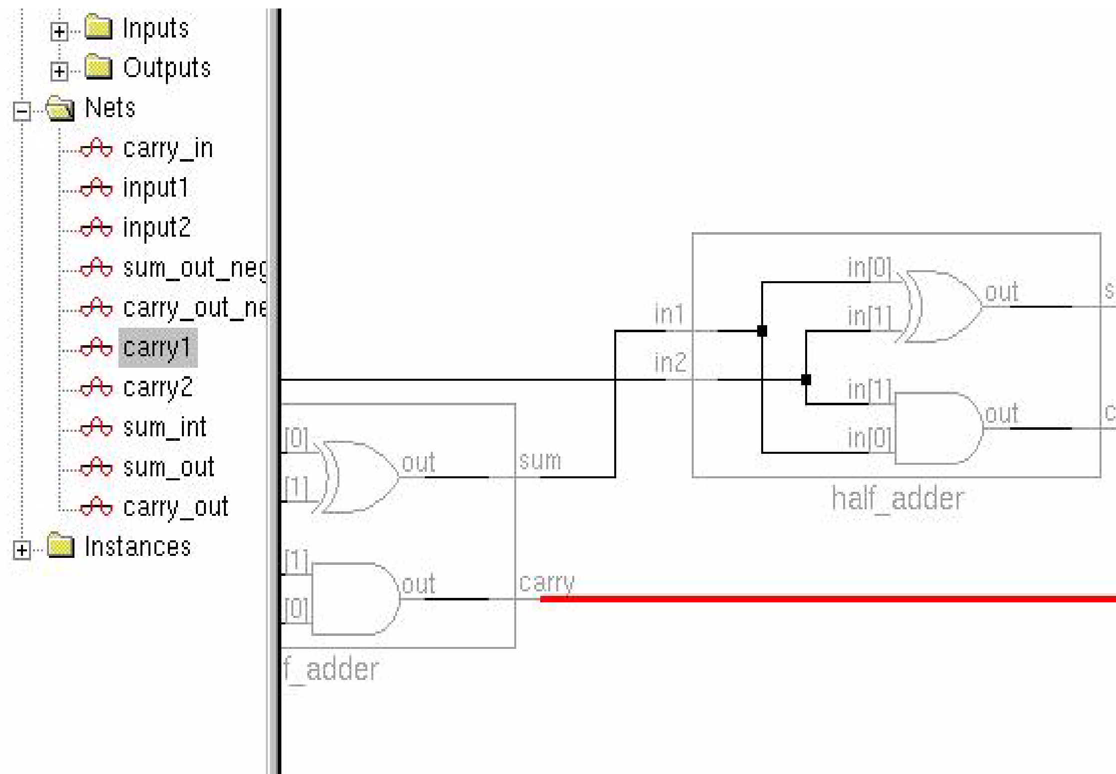


Figure 18: RTL Schematic highlighting a chosen net.

(12) If your design is hierarchical (i.e. it makes us of components and port map statements), you may view the hierarchy within the schematic by moving the cursor to an empty area of the schematic and right-clicking with the mouse button and selecting the Show Hierarchy item. You may print your schematic to a Postscript file by selecting the yellow printer icon located top left portion of the main window:

In the Print form, specify Print to File and select OK as shown in Figure 19. A Print to File form will appear, specify location and a filename such as ‘Full_Adder_Schematic’ and click on the Save button. A file called Full_Adder_Schematic.prn will be created in the location you specified. This is a Postscript file which may be printed to a laser printer with the ‘lpr -P printer_name’ command. Figure 20 is the schematic diagram of the full-adder circuit showing the design hierarchy.

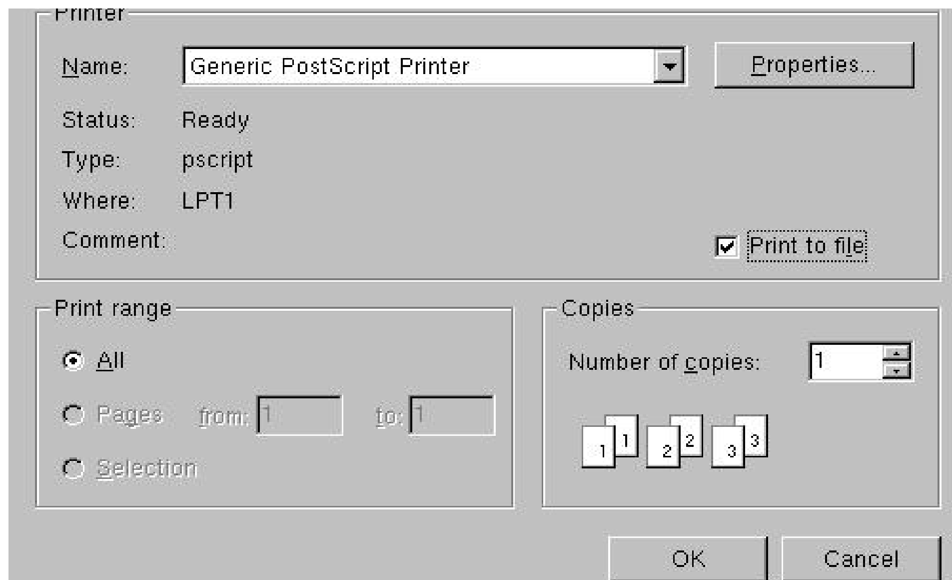


Figure 19: Print to File form.

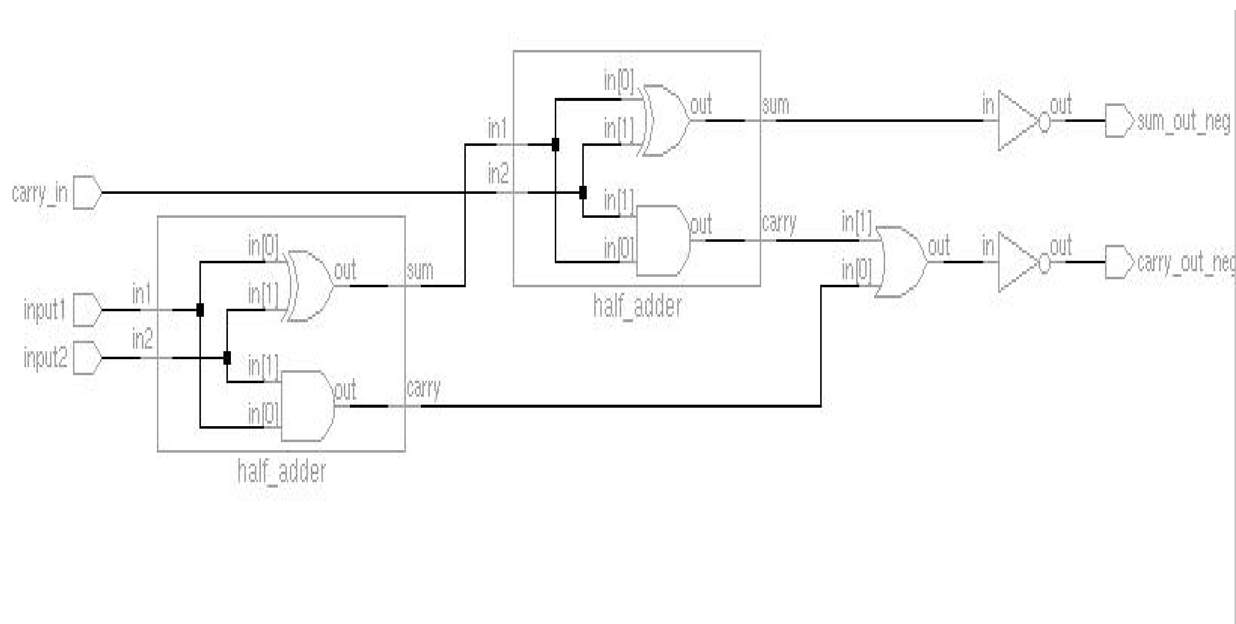


Figure 20: RTL Schematic of Full Adder circuit showing hierarchy.

PART III : Implementation using Xilinx ISE

The end result of the steps performed in the previous section was the creation of a netlist file in a format known as Electronic Design Interchange Format (EDIF). An EDIF file is a netlist of basic logic gates. The Xilinx ISE tools use EDIF as one possible input format. The steps involved to arrive at a functioning implementation beginning with an EDIF file are summarized below:

(i) the EDIF file is converted into a netlist of Xilinx Logic Cells. This step is referred to as **technology mapping** or **partitioning**. The mapping also attempts to perform some optimization either in terms of the number of Logic Cells required or timing requirements.

(ii) the next step is to **place** each of the Logic Cells generated from the mapping phase into a specific location within the target FPGA. Once the Logic Cells have been placed, they must be interconnected using the available wiring resources and switches within the FPGA. This is referred to as **routing**.

(iii) once a design has been placed and routed, a **configuration** file is created which is used to program the FPGA. The Xilinx CAD tools will create a file with a .bit extension. This file is then used to generate a configuration file using the Xilinx Impact tool.

I. Setting up the user environment to run the Xilinx ISE program

Prior to running the Xilinx tools, it is necessary to setup the Linux environment to run the Xilinx tools. Type the following from the Linux prompt:

```
ted@acmilan 12:58pm >  source   /CMC/ENVIRONMENT/xilinx.env
```

Note that the above *xilinx.env* file is actually a symbolic link pointing to /CMC/ENVIRONMENT/xilinx_9.2i.env. The symbolic link *xilinx.env* points to the currently installed and supported version of the Xilinx tools. Occasionally, there may co-exist other versions. Graduate student researchers may need to run other versions, they should refer to the .env files found in the directory /CMC/ENVIRONMENT and source the appropriate file (if it exists).

II. Implementing a Design with the Xilinx ISE Project Navigator

(1) create a subdirectory called Xilinx from within your Modelsim diretory. This directory will be used to hold the intermediate files produced by the Xilinx tools. The .bit file created during the configuration step will also be saved within the structure of this directory.

(2) Place and route is performed with the ISE software. To start the ISE software type ise from the Linux prompt (the & symbol after the command name will cause the tool to run in the background and you will be returned back to the Linux prompt after the window appears)

```
ted@deadflowers Xilinx 2:35pm >  ise  &
```

(3) The Xilinx Project Navigator window will appear as shown in Figure 21. A Tip of the Day window will also appear, select OK in it to close the Tip of the Day.

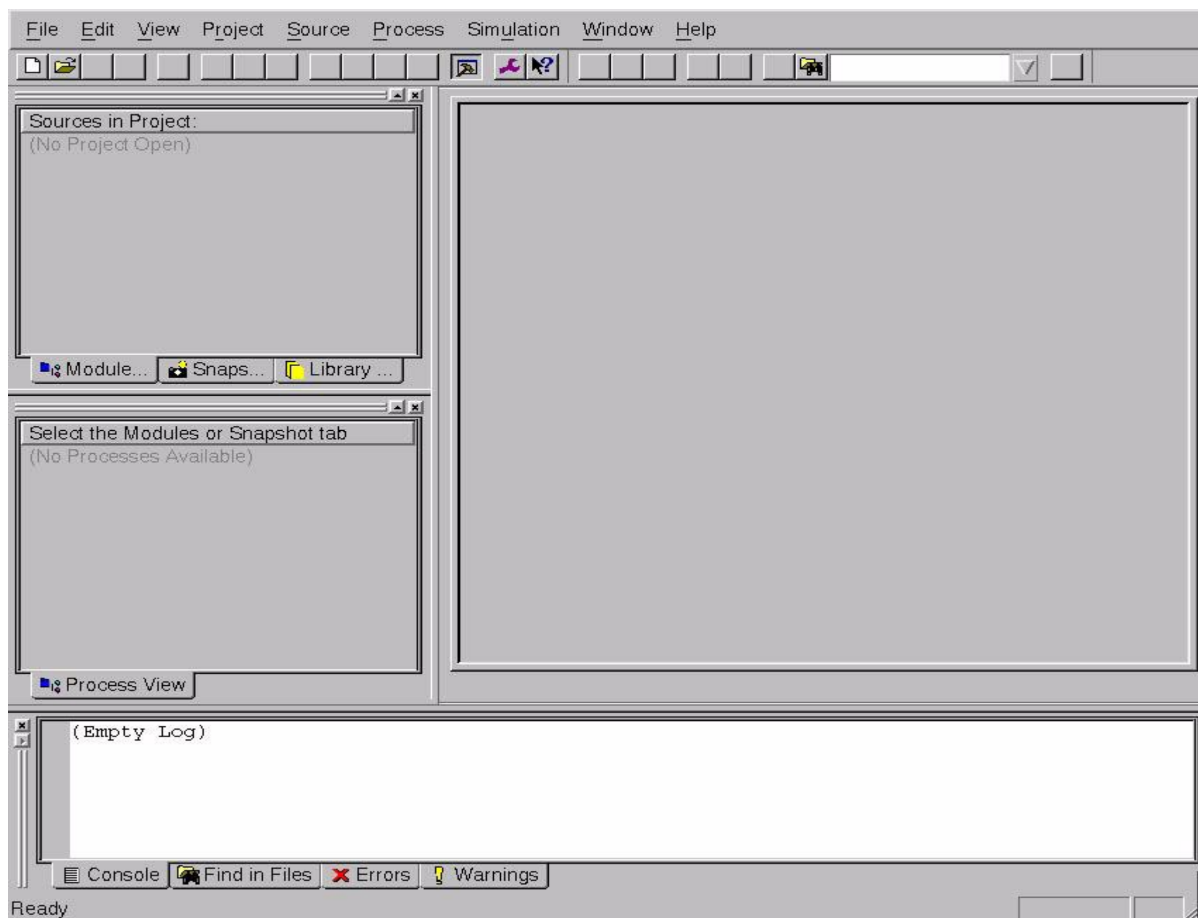


Figure 21: Xilinx Project Navigator window.

Select **File -> New Project** from the list of items located across the top of this window. The **New Project** window will appear. Specify the following in the **New Project** window:

Project Name: Full_Adder
Project Location: specify the full path to your Xilinx directory, for example:
/nfs/home/t/ted/Modelsim/Xilinx/Full_Adder
Top-Level Module: select EDIF from the list.

Select **Next>** when you have entered the values. Refer to Figure 22 for the details of this form.

Figure 22: New Project window.

(4) You will need to create a User Constraints File in order for the Xilinx ISE tools to associate input/output ports in your design with physical pin numbers on the FPGA chip. For the purpose of this tutorial, create a text file called full_adder_negated.ucf file in the same directory that the Precision RTL tool generated your .edf (EDIF netlist) file. The contents of this full_adder_ne gated.ucf file should be:

```
CONFIG STEPPING="0";
NET carry_in LOC  = AC11;
NET input1  LOC = AD11;
NET input2  LOC = AF8;
NET carry_out_neg   LOC = AC4;
NET sum_out_neg LOC =  AC3;
```

Usually, such a constraints file is created **before** one begins implementation with the ISE tools. The .ucf file tells the ISE implementationt tools that the carry_in input should be mapped to I/O pin AC11 of the FPGA device. This pin is connected to a user DIP switch on the board. Similarly, the two outputs of the full_adder (carry_out_neg , sum_out_neg) are mapped to pins AC4 and AC3 which are connected to LEDs on the download board.

Once you have selected **Next>**, the New Project form will change and you will be prompted to enter the Input File and the User Constraint File. Specify your full_adder_negated.edf file and the

full_adder_negated.ucf file you have created. You may use the **...** buttons to the right of each field to browse and select a particular file from your directory structure. Select **Next>** when you are done. Refer to Figure 23.

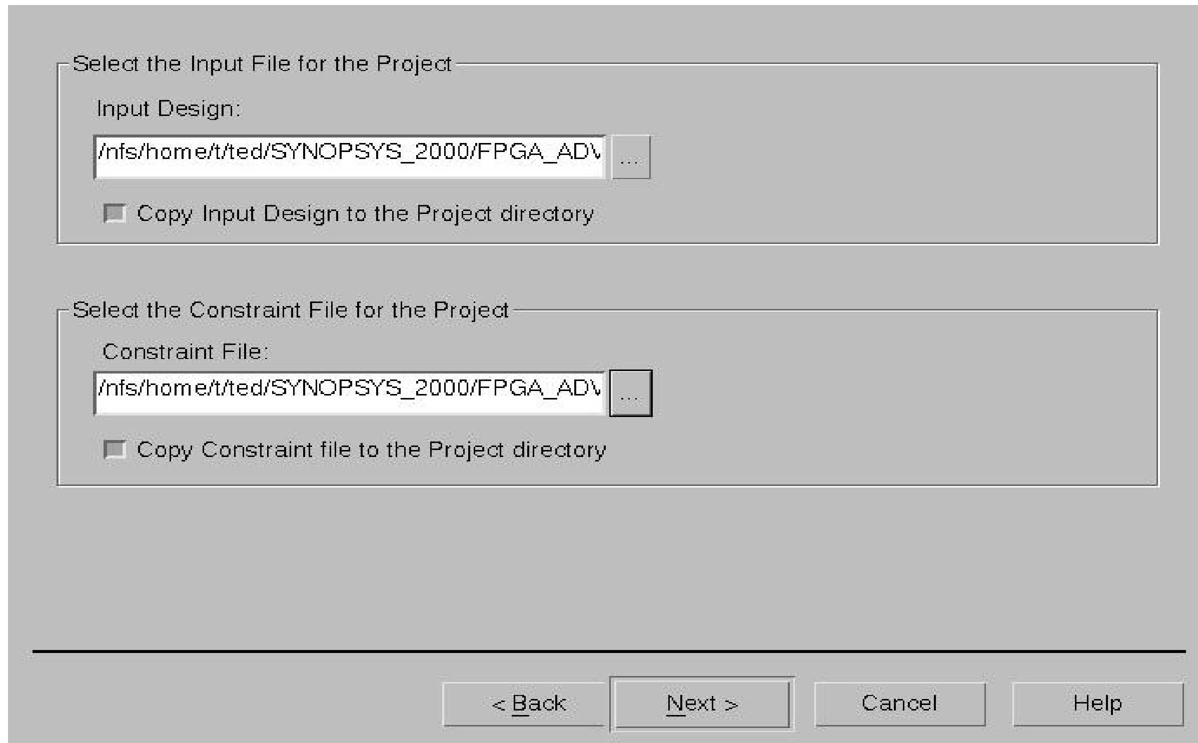


Figure 23: Specifying the Input File and the User Constraints File.

(5) You must now specify whcih device you use to use. Refer to Figure 24 and specify the following in the form:

Device Family: Virtex2P
Device: xc2vp30
Package: ff896
Speed Grade: -7
Top_level Module Type: EDIF
Simulator: Other

Select **Next>**.

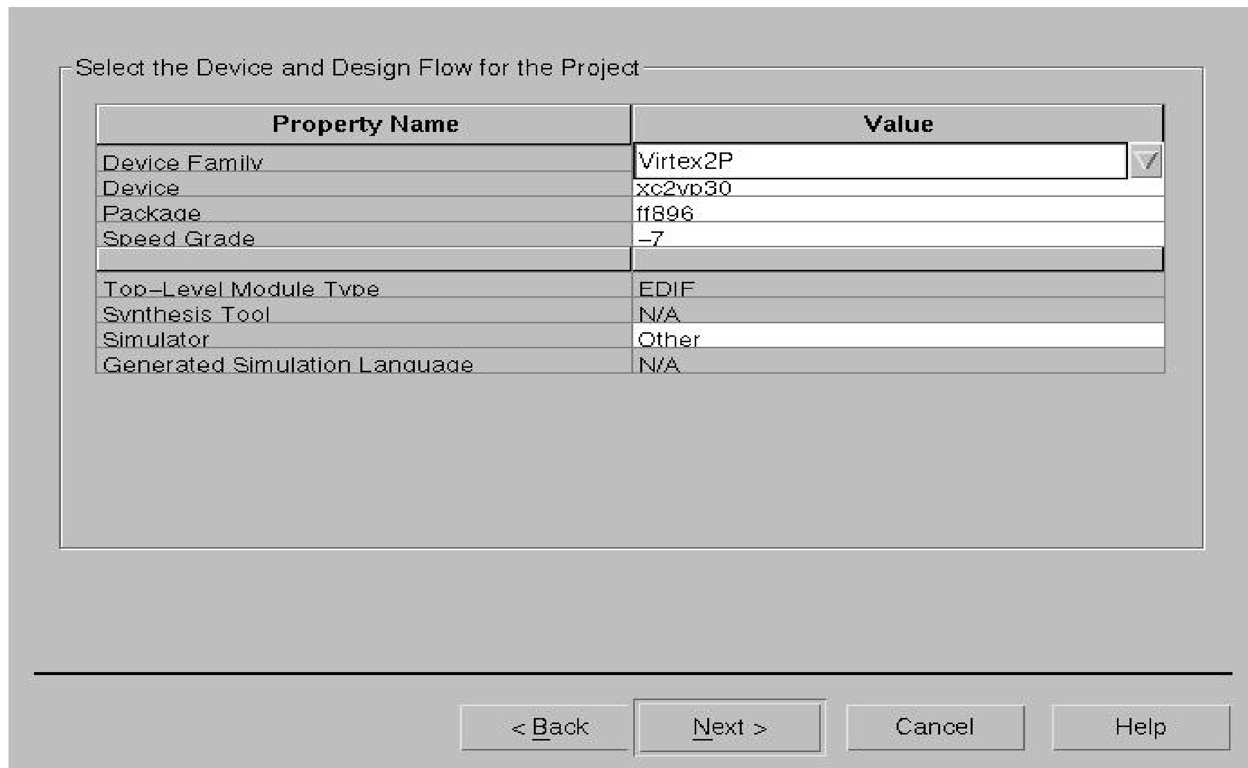


Figure 24: Specifying the Device.

(6) In the New Project Information window select **Finish**. See Figure 25.

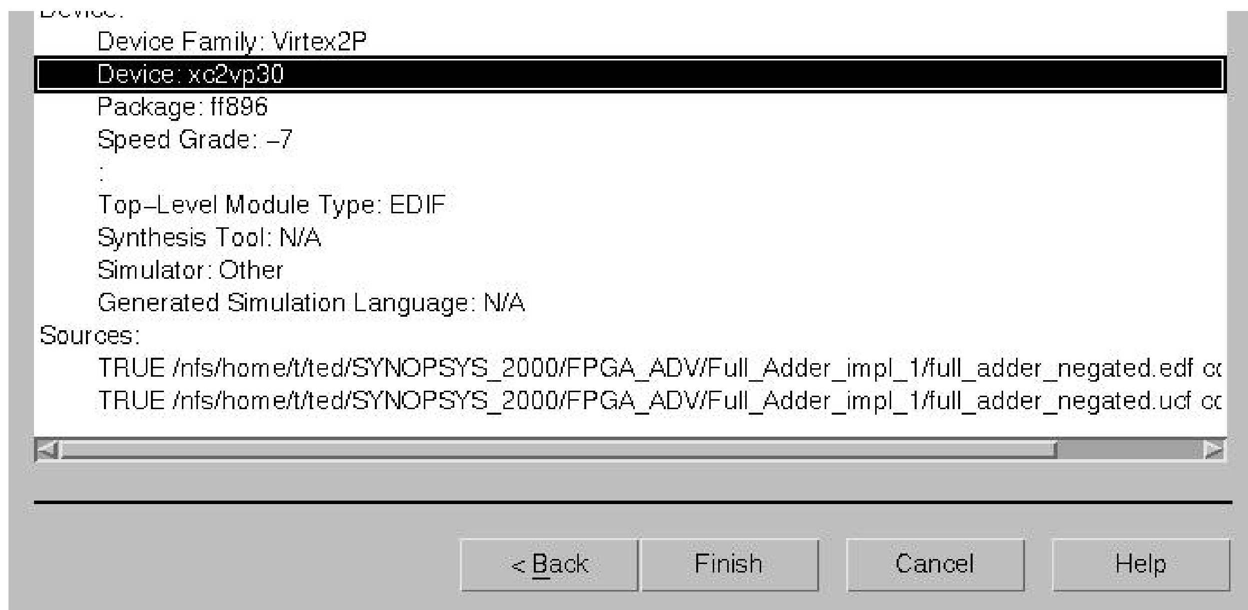


Figure 25: New Project Information window.

(7) The Project Navigator window will change to that shown in Figure 26.

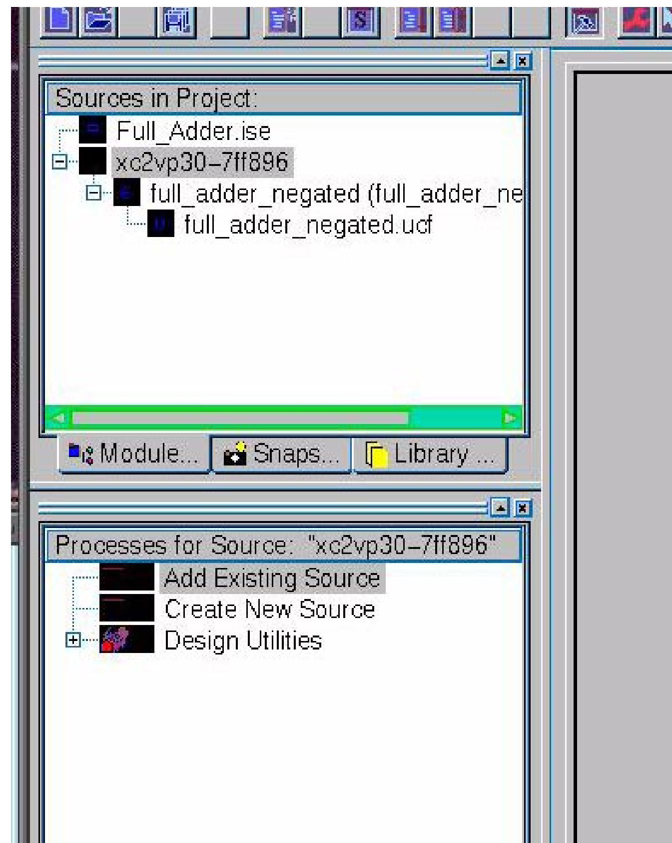


Figure 26.

(8) Select the full_adder_negated.edf file (the file listed beneath xc2vp30-7ff896). You will now note that the **Processes for Source: "xc2vp30-7ff896"** have changed to include the "Implement Design" as shown in Figure 27.

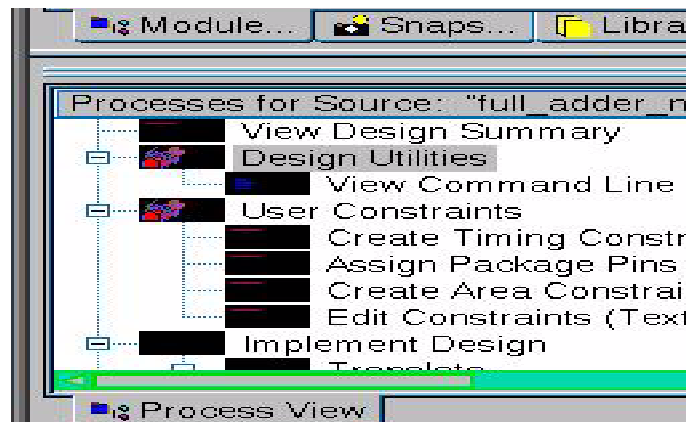


Figure 27: Process including Implement Design.

To start the Implement Design process, double click on Implement Design. Messages indicating

the progress of the process will be displayed in the bottom pane of the Project Navigator. These messages are also written to a log file (__projnav.log) in your specified Xilinx Project directory (i.e. /nfs/home/t/ted/Modelsim/Xilinx/Full_Adder). If there are any errors or warnings, you should consult the log file to determine their cause. Most warnings can be safely ignored, others merit futher investigation and removal. Incidentally, the entire Xilinx ISE implementation flow can be executed using a command-line interface, rather then the GUI method illustrated in this tutorial. One can even create a shell script consisting of the appropriate commands and execute this command in the background. This can be useful for large designs which may take up to several hours/days to implement. Essentially, the log file indicates the commands and their agruments as in:

```
ngdbuild -intstyle ise -dd
/nfs/home/t/ted/Modelsim/Xilinx/Full_Adder/_ngo -uc full_adder_negated.ucf -p
xc2vp30-ff896-7 full_adder_negated.edf full_adder_negated.ngd

edif2ngd -quiet "full_adder_negated.edf" "_ngo/full_adder_negated.ngo"

/CMC/tools/xilinx_7.1i/bin/sol/map -ise
/nfs/home/t/ted/Modelsim/Xilinx/Full_Adder/Full_Adder.ise -intstyle ise -p
xc2vp30-ff896-7 -cm area -pr b -k 4 -c 100 -tx off -o
full_adder_negated_map.ncd full_adder_negated.ngd full_adder_negated.pcf

par -w -intstyle ise -ol std -t 1 full_adder_negated_map.ncd
full_adder_negated.ncd full_adder_negated.pcf
```

The actual details may vary, but this should be a sufficient starting point for the UNIX gurus to implement their design using the command line interface.

The Xilinx Project directory also contains a number of useful reports giving the device resource utilization and timing values. For example, the file full_adder_negated.twr gives the following (useful) information:

ted@acmilan Full_Adder 11:16am >more full_adder_negated.twr

```
--------------------------------------------------------------------------------
Release 7.1i Trace H.38
Copyright (c) 1995-2005 Xilinx, Inc.  All rights reserved.

/CMC/tools/xilinx_7.1i/bin/sol/trce -ise
/nfs/home/t/ted/SYNOPSYS_2000/Xilinx/Full_Adder/Full_Adder.ise -intstyle ise -
e
3 -l 3 -s 7 -xml full_adder_negated full_adder_negated.ncd -o
full_adder_negated.twr full_adder_negated.pcf


Design file:              full_adder_negated.ncd
Physical constraint file: full_adder_negated.pcf
Device,speed:             xc2vp30,-7 (PRODUCTION 1.90 2005-01-22)
Report level:             error report
```

```
Environment Variable      Effect
--------------------      ------
NONE                      No environment variables were set
------------------------------------------------------------------------------
--

INFO:Timing:2698 - No timing constraints found, doing default enumeration.
INFO:Timing:2752 - To get complete path coverage, use the unconstrained paths
   option. All paths that are not constrained will be reported in the
   unconstrained paths section(s) of the report.

Data Sheet report:
-----------------
All values displayed in nanoseconds (ns)

Pad to Pad
---------------+---------------+---------+
Source Pad     |Destination Pad|  Delay  |
---------------+---------------+---------+
carry_in       |carry_out_neg  |    6.044|
carry_in       |sum_out_neg    |    5.835|
input1         |carry_out_neg  |    5.607|
input1         |sum_out_neg    |    5.981|
input2         |carry_out_neg  |    5.654|
input2         |sum_out_neg    |    5.839|
---------------+---------------+---------+
```

The full_adder_negated.mrp file reports the amount of FPGA resources used to implement the design:

ted@acmilan Full_Adder 11:21am >more full_adder_negated.mrp

```
Release 7.1i Map H.38
Xilinx Mapping Report File for Design 'full_adder_negated'

Design Information
------------------
Command Line   : /CMC/tools/xilinx_7.1i/bin/sol/map -ise
/nfs/home/t/ted/SYNOPSYS_2000/Xilinx/Full_Adder/Full_Adder.ise -intstyle ise -
p
xc2vp30-ff896-7 -cm area -pr b -k 4 -c 100 -tx off -o
full_adder_negated_map.ncd
full_adder_negated.ngd full_adder_negated.pcf
Target Device  : xc2vp30
Target Package : ff896
Target Speed   : -7
Stepping Level : 0
Mapper Version : virtex2p -- $Revision: 1.26.6.3 $
Mapped Date    : Tue Jun 27 11:00:56 2006
Design Summary
--------------
```

```
Number of errors:      0
Number of warnings:    1
Logic Utilization:
  Number of 4 input LUTs:              2 out of  27,392    1%
Logic Distribution:
  Number of occupied Slices:           1 out of  13,696    1%
  Number of Slices containing only related logic:       1 out of       1  100%
  Number of Slices containing unrelated logic:          0 out of       1    0%
    *See NOTES below for an explanation of the effects of unrelated logic
Total Number 4 input LUTs:             2 out of  27,392    1%

  Number of bonded IOBs:               5 out of     556    1%
  Number of PPC405s:                  0 out of       2    0%
  Number of GTs:                       0 out of       8    0%
  Number of GT10s:                     0 out of       0    0%

Total equivalent gate count for design:  12
Additional JTAG gate count for IOBs:  240
Peak Memory Usage:  166 MB
```

The full_adder_negated.pcf (Physical Constraints File, not **Physical Graffitti** - a **Led Zeppelin** album title) lists the inputs/outputs and the physical pinds they are associated with. Recall that the .ucf file originally specified these locations.

ted@acmilan Full_Adder 11:23am >more full_adder_negated.pcf

```
//! ************************************************************************
// Written by: Map H.38 on Tue Jun 27 11:01:08 2006
//! ************************************************************************

SCHEMATIC START;
COMP "carry_out_neg" LOCATE = SITE "AC4" LEVEL 1;
COMP "sum_out_neg" LOCATE = SITE "AC3" LEVEL 1;
COMP "carry_in" LOCATE = SITE "AC11" LEVEL 1;
COMP "input1" LOCATE = SITE "AD11" LEVEL 1;
COMP "input2" LOCATE = SITE "AF8" LEVEL 1;
SCHEMATIC END;
```

(9) The next step is to generate a .bit file. This is a special file which is used to program (configure) the FPGA. A .bit file may be downloaded directly to a FPGA board using a special communications cable (MultiLinx cable), or the .bit file may be used to produce a configuration file which may be copied onto a Compact Flash card. This tutorial will explain the use of programming the board using the System Ace Compact Flash method.

VERY IMPORTANT! PAY YOUR ATTENTION! READ THIS FIRST !!!!!!

Prior to generation of the .bit file, JTAG clock **must** be selected as the Startup Clock (see page 3-3 of Impact User's Guide). Failure to specify JTAG CLK as the Startup Clock will result in an error during the download of the configuration file to the board. See Impact User's Guide page 5-

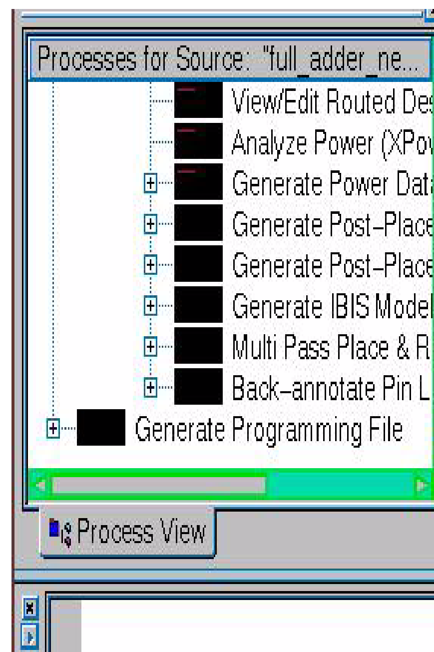
6 :

"NOTE: THE SAME STARTUP CLOCK RESTRICTIONS THAT APPLY FOR CONFIGURING DEVICES WITH A CABLE ALSO APPLY TO ADDING BITSTREAMS TO THE SYSTEM ACE FILES. FOR SYSTEM ACE CF, ONLY BITSTREAMS WITH STARTUP CLOCK SETTINGS OF BOUNDARY SCAN (JTAG) CLOCK ARE ALLOWED"

To specify JTAG clock as the Startup Clock select:

Generate Programming File

from the Processes for Current Source in the Project Navigator window so that it becomes highlighted as shown below:



Next, select **<u>Process</u>** from the Project Navigator, this will cause a popup menu to appear listing the following choices:

Run
Rerun
RerunAll
Properties...

Select **Properties...** from this list. The Process Properties window will appear as shown in Figure 28. Select the **Startup Options** button located at the top left-hand side.

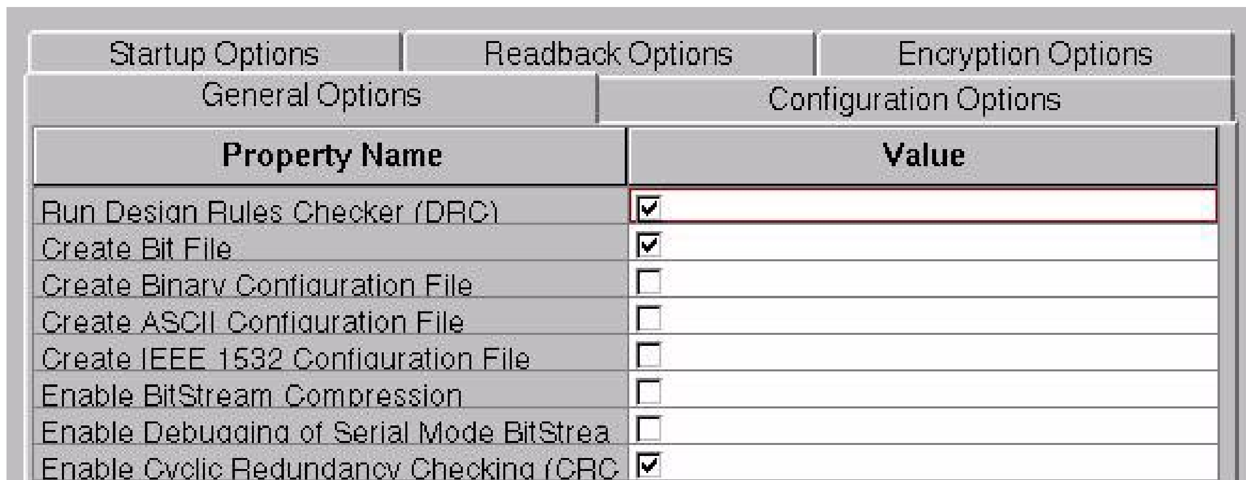


Figure 28: Process Properties window.

Next, select JTAG clock as the FPGA Startup Clock as shown in Figure 29 and click **OK**.

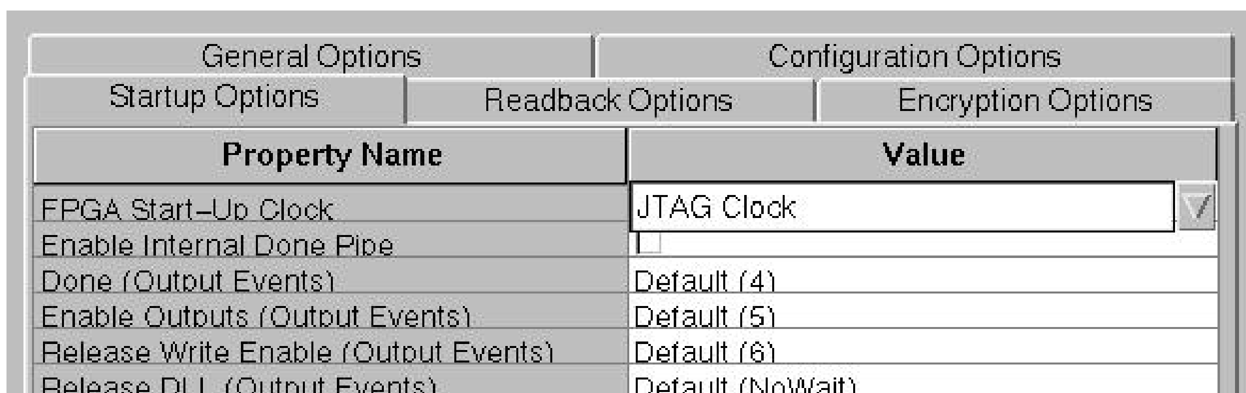


Figure 29: Specifying JTAG Clock as the Startup-Clock.

Once this Startup Clock has been specified, double click on **Generate Programming File** to generate the full_adder_negated.bit file. The file size should be similar to:

```
ted@acmilan Full_Adder 11:30am >ls -al *.bit
-rw-------   1 ted      ted       1448824 Jun 27 11:56 full_adder_negated.bit
```

You may now select **File -> Save All** from the Project Navigator and then **Exit** the application.

III. Using Xilinx Impact to program a Compact Flash card with a System ACE File

The Xilinx Impact software will now be used to generate a System ACE file which can be copied to a Compact Flash card. The System ACE file contains all the information needed to program the FPGA device. It has the added benefit of being non-volatile. The default configuration mode of the demonstration boards is such that the FPGA configures itself from the Compact Flash card upon power-up.

(1) Source the /CMC/ENVIRONMENT/xilinx.env file if you have not already done so.

ted@deadflowers Xilinx 3:29pm >source /CMC/ENVIRONMENT/xilinx.env

(2) Change into your Xilinx/Full_adder directory and start the Impact software:

ted@deadflowers Xilinx 2:59pm >cd Full_Adder/
ted@deadflowers Full_Adder 2:59pm >impact &

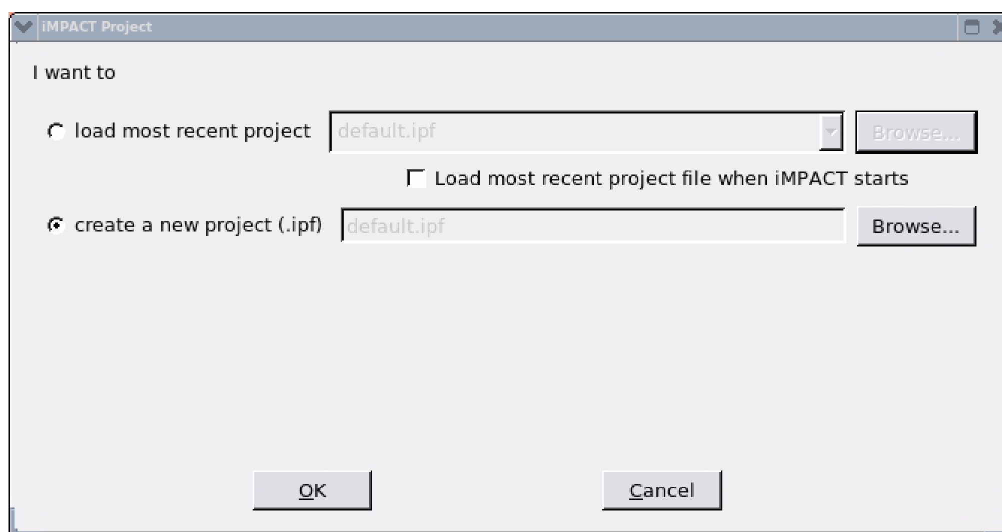
Select OK in the "The Project Directory / is either not writeable or does not exist. iMPACT has changed the Project Directory to the current working directory ..." message window if it appears.

(3) In the Impact Project window, select

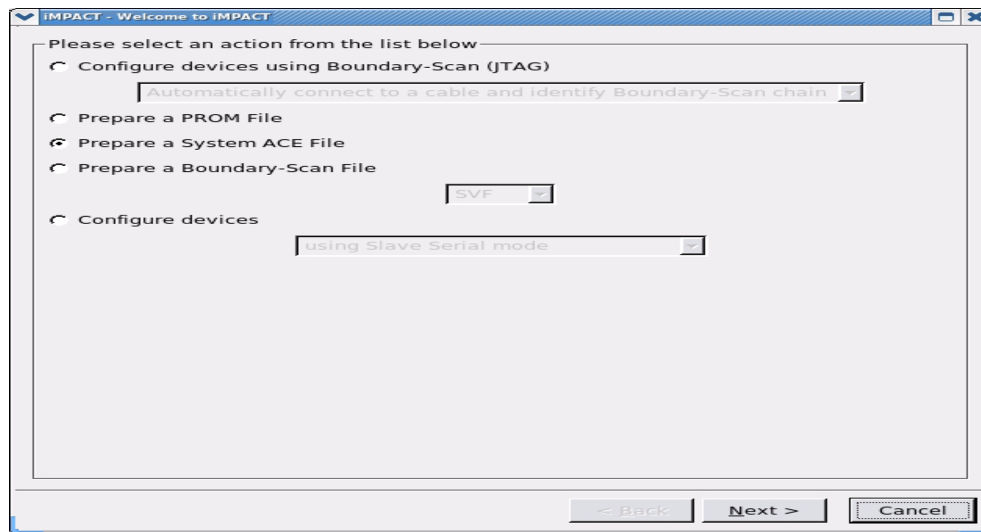
I want to

• create a new project (.ipf) default.ipf

and Select OK.



(4) Select : Prepare a System ACE file and select Next in the Welcome to iMPACT window.



(5) Choose Novice as the Operating Mode and select Next.

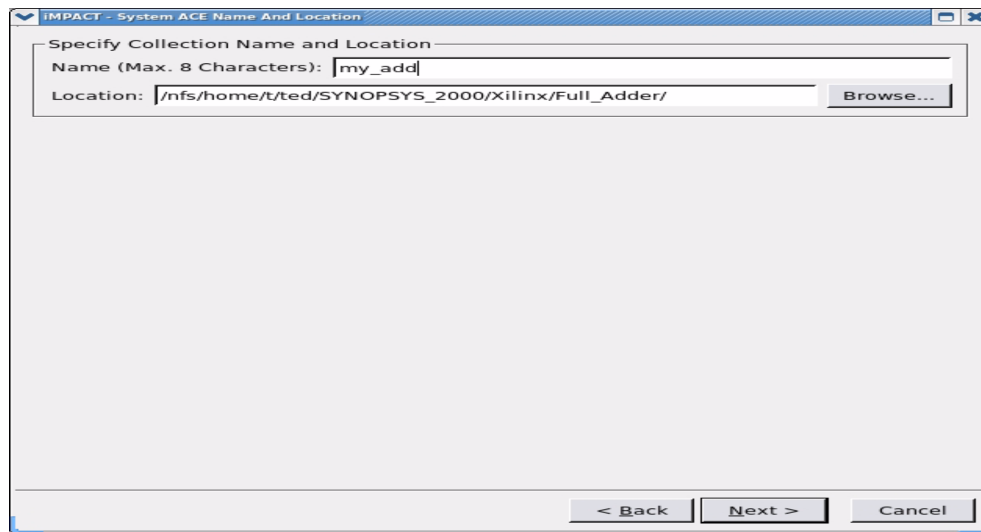
(6) Select System ACE CF Size

Size : 128 MBits

and select Next

(7) Specify Collection Name and Location:

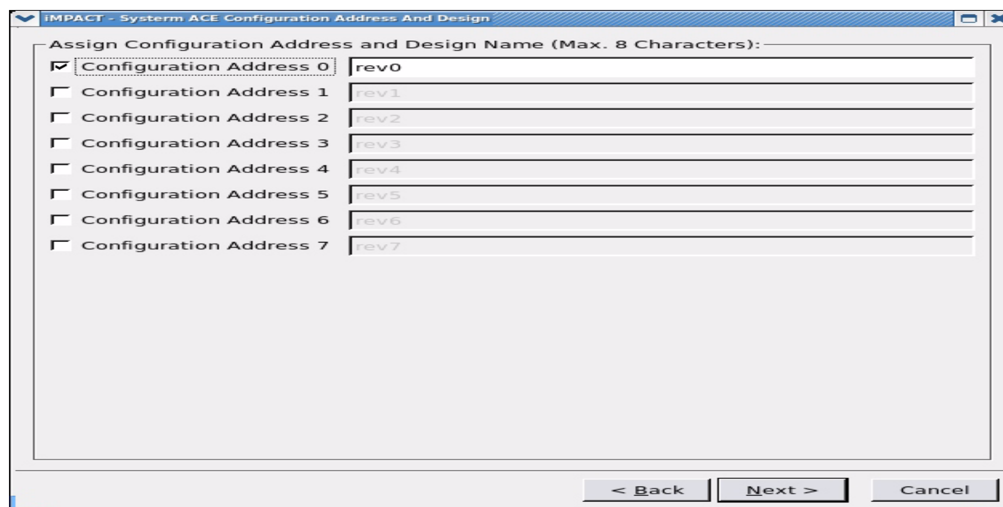
Name : Full_Add (give some nice meaningful name)
Location: it will have the path to your Full_Adder directory



(8) Assign Configuration Address And Design Name ...

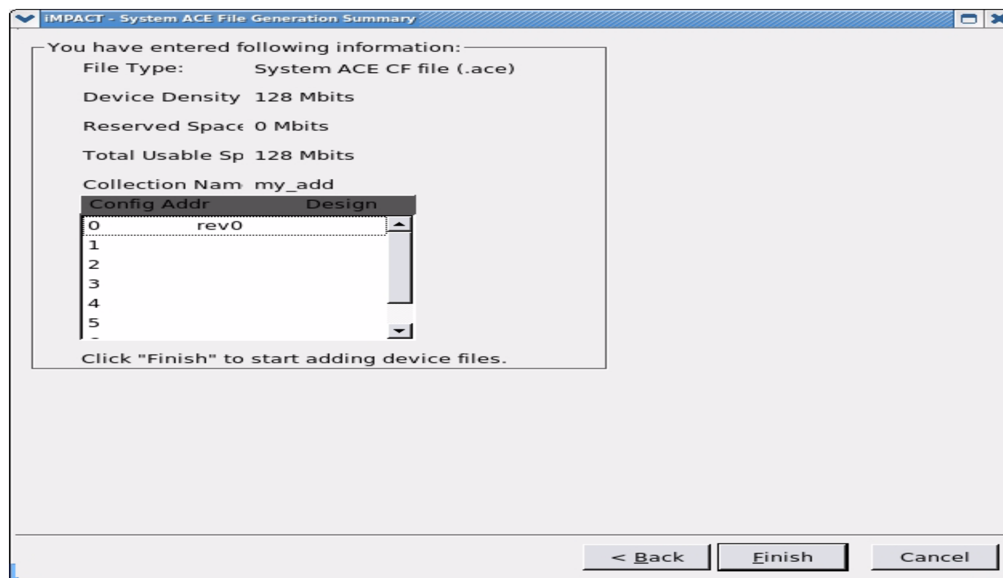
Select the Configuration Address 0 tick box.

.



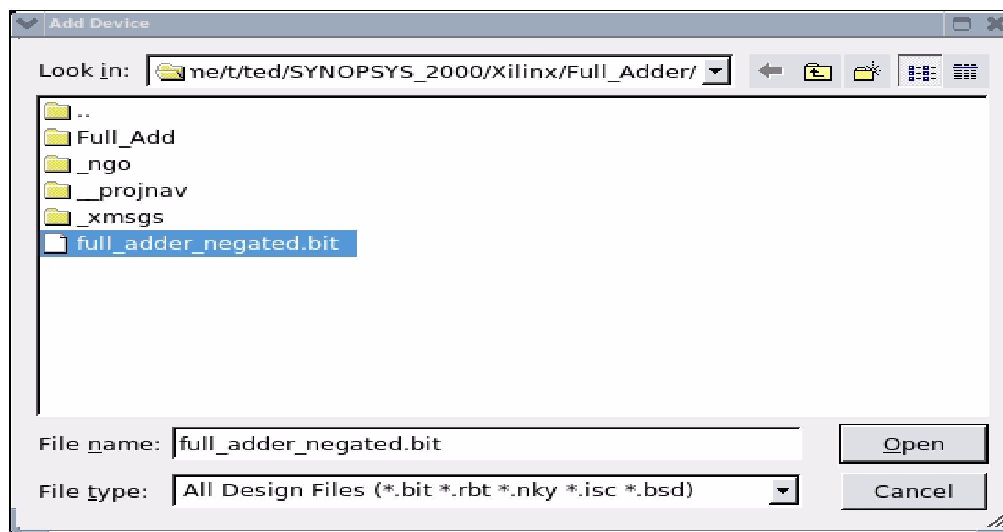
Select Next

(9) Select Finish in the System ACE file Generation Summary

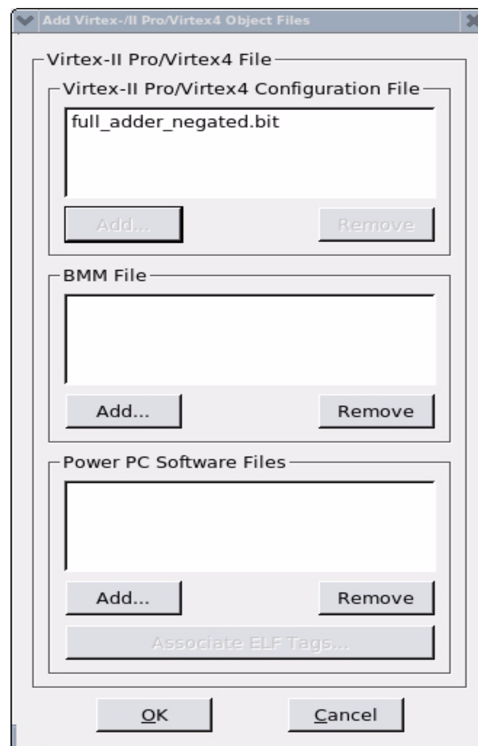


(10) Select OK in the “Now sart assigning device file to Config Address:0 dialog box.

(11) Select your full_adder_negated.bit file in the Add Device window and select OPEN

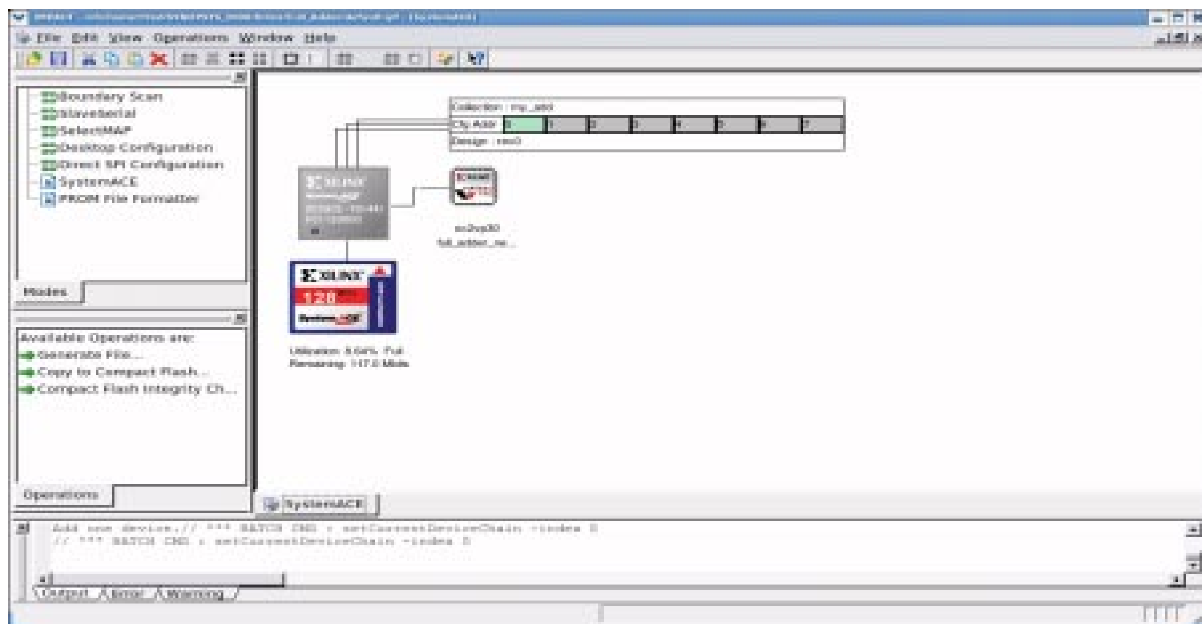


(12) Select OK in the Add Virtex-II Pro/Virtex4 Object file window.

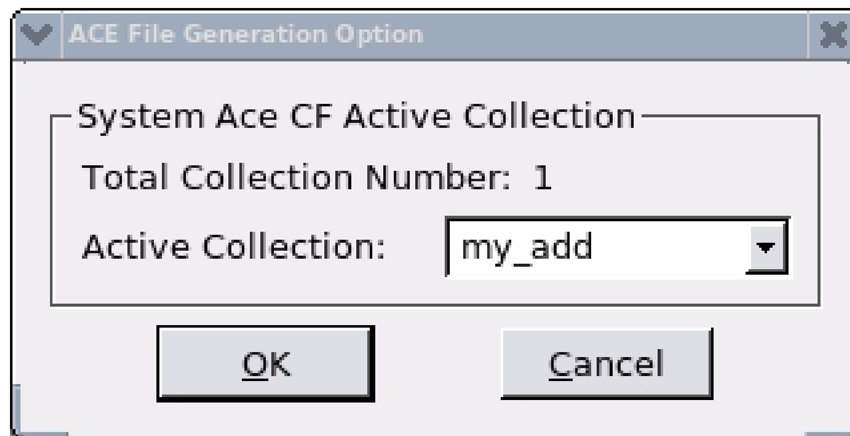


(13) Answer No to the “Would you like to add another design file to Config Address:0 ? dialog box.

(14) Select the ==> Generate File in the bottom left portion of the main Impact Window.



(15) Select OK in the ACE file Generation Option



The ACE file Generation Successful message will be displayed in the main Impact window. Select File -> Save Project and then File -> Exit.

(16) Examine the contents of the directory you specified in step 7:

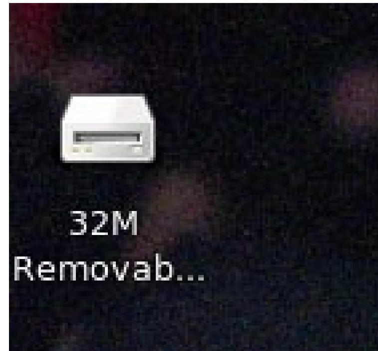
```
ted@acmilan Full_Add 12:56pm >pwd
/nfs/home/t/ted/SYNOPSYS_2000/Xilinx/Full_Adder/Full_Add
ted@acmilan Full_Add 12:56pm >ls -al
total 16
drwx------   3 ted     ted        4096 Jun 27 12:26 .
drwx------   2 ted     ted        4096 Jun 27 12:26 rev0
-rw-------   1 ted     ted          81 Jun 27 12:26 xilinx.sys

ted@acmilan Full_Add 12:56pm >cd rev0
ted@acmilan rev0 12:57pm >ls -al
total 1428
drwx------   2 ted     ted        4096 Jun 27 12:26 .
drwx------   3 ted     ted        4096 Jun 27 12:26 ..
-rw-------   1 ted     ted     1449797 Jun 27 12:26 rev0.ace
```

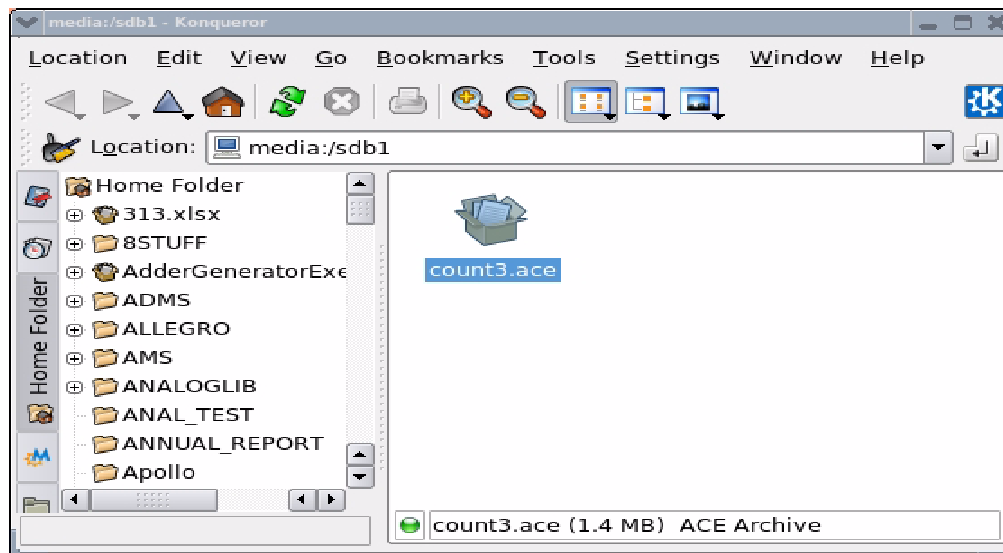
The default name of the SystemAce file that Xilinx creates is “rev0.ace”. If you wish, you may copy it to some other file name such as “full_adder.ace” . This is the file which is to be used to program the FPGA.

(17) Copy the rev0.ace to the Compact Flash card and plug the card into the board . The board will configure itself from the compact flash card (ensure that there is only one .ace file stored in the card). The DONE LED on the development board should light up when the FPGA has been configured. If there is an error during the programming of the FPGA device, ask your lab instructor to verify the postion of the DIP switches on the FPGA board.

(18) Under Linux, the filesystem for the removable Compact flash card will be automounted only after the icon representing it on the KDE desktop has been selected:



Use the mouse to select this icon (by double clicking) and the filesystem called /media/disk will be mounted. In addition, a Media window will open:



The df command may be used to verify that the /media/disk filesystem has been mounted:

```
ted@deadflowers ~ 6:04pm >df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda2              9920624   3820496   5588060  41% /
/dev/shm               1782792        12   1782780   1% /dev/shm
filer-software:/vol/sw_cmc
                     1717986944 294931136 1423055808  18% /nfs/sw_cmc
```

```
filer-users:/vol/users/users_unix
                     1760285184 958833056 801452128  55% /nfs/home
filer-software:/vol/software/software/encs
                     742391808 439336384 303055424  60% /nfs/encs
/dev/sdb1                31202      7080     24122  23% /media/disk
```

One may copy the “rev0.ace” (any other file) to the Compact flash card using the cp command as in:

ted@deadflowers rev0 6:08pm >cp rev0.ace /media/disk

Use the ls command to veirfy that the file has been copied:

ted@deadflowers rev0 6:08pm >ls -al /media/disk
total 2856
drwxr-xr-x 2 ted root   16384 Jul 21 18:08 .
drwxr-xr-x 4 root root   4096 Jul 21 17:59 ..
-rwxr-xr-x 1 ted root 1449797 Jul 21 18:08 rev0.ace

IMPORTANT: After having copied the System Ace file to to compact flash card, it is necessary to use the ‘sync’command to flush to file buffer.

ted@deadflowers rev0 6:09pm > sync

After the ’sync’, it is now safe to remove the CF card from the reader to program the FPGA board. If you don’t ’sync’ after writing to the card, the file size on the CF card will be 0 bytes, and there will be a SystemAce Error on the FPGA board.

PART IV : Xilinx FPGA Development Board

The Xilinx University Program Virtex-II Pro development board contains a Virtex-II Pro XC2VP30 FPGA device in an FF896 BGA (Ball Grid Array) package. This FPGA device has the equivalent logic capability of approximately 30 000 000 logic gates. It contains 13 969 slices (a slice contains a RAM look-up table which is used to implement combinational logic, a slice also contains dedicated flip-flops for sequential logic implementation), 428 Kb (kilobits) of distributed RAM, over 2000 Kb of Block RAM, and 136 multipliers (18 bit x 18 bit). Figure 30 is a top view photo of the XUP Virtex-II Pro development system.

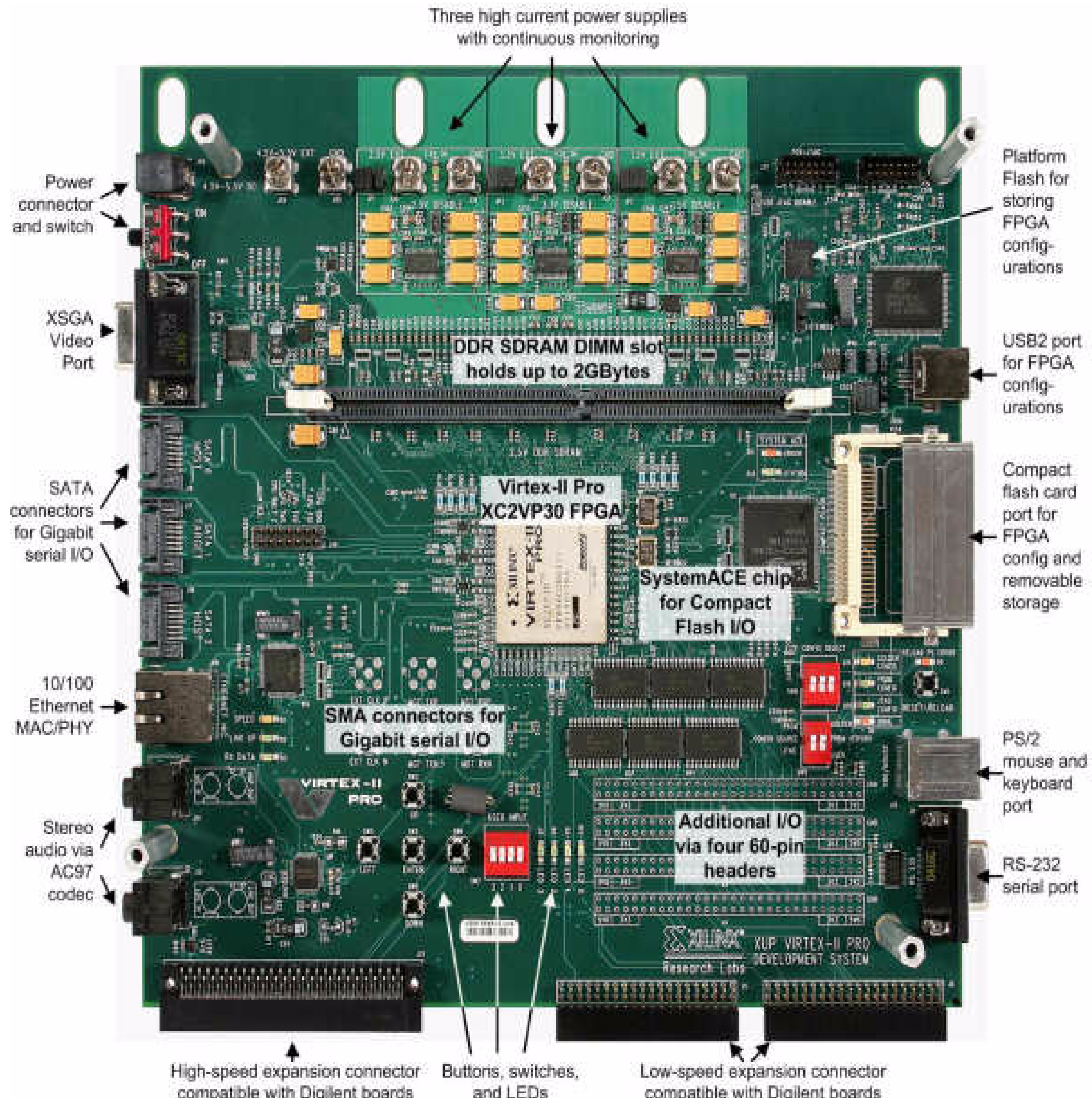


Figure 30: XUP Virtex-II Pro development system[3].

User LEDS and Switches

The XUP Virtex-II Pro development board contains four user-defined LEDs as well as four DIP switches and five pushbutton switches. Note that none of the switches are debounced. The LEDs are **active LOW**. Table 1 provides the connections between these LEDs and switches and the FPGA device.

Table 1: User LEDs and Switch Connections

Device	FPGA Pin
LED_0	AC4
LED_1	AC3
LED_2	AA6
LED_3	AA5
SW_0	AC11
SW_1	AD11
SW_2	AF8
SW_3	AF9
PB_ENTER	AG5
PB_UP	AH4
PB_DOWN	AG3
PB_LEFT	AH1
PB_RIGHT	AH2

The 4 DIP switches (SW_0 - SW_3 in Table 1) produce a **logic-0** value when placed in the **UP** position. The 5 momentary contact pushbutton switches produce a logic-0 when they are pressed upon, otherwise they produce a logic-1 value. Thus, they are useful for active low reset inputs. The 4 LEDs (LED_0 - LED_3) are **active LOW**, this means that the LED will light up when driven by a logic-0 signal.

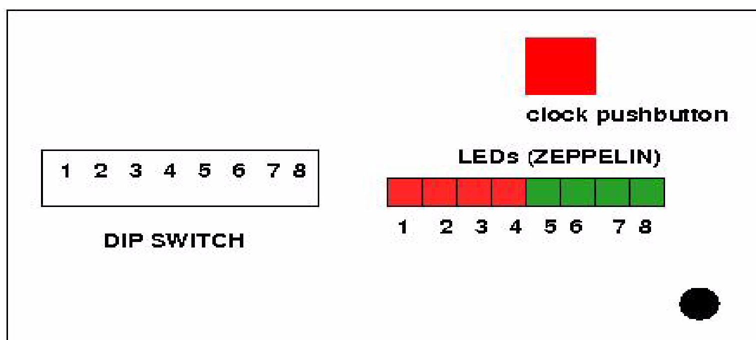
Expansion LEDs and DIP Switches

Due to the limited number of available user LEDs and switches on the XUP Virtex-II board, an expansion input/output module was added to the develoment board by our ECE technical team. This expansion module consists of a debounced clock implemented with a 555 timer integrated circuit, 8 dual inline pin (DIP) switches, and 8 LEDs. Table 2 lists the connections between the expansion IO and the FPGA device.

Table 2: Expansion IO Connections

Device	FPGA Pin
555 timer output (clock)	T4
SW_1	N5
SW_2	L4
SW_3	N2
SW_4	R9
SW_5	M3
SW_6	P1
SW_7	P7
SW_8	N3
LED1	P2
LED2	R7
LED3	P4
LED4	T2
LED5	R5
LED6	R3
LED7	V1
LED8	T6

Note that the expansion module's LEDs are **active LOW**. Refer to Figure 31 for the numbering of the 8 switches, 8 LEDs, and the location of the clock pushbutton switch, as well as the operation of the DIP switches.



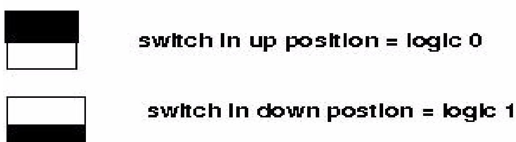


Figure 31: Expansion IO module switch and LED locations.

Board Documentation

Complete documentation and schematics for the XUP Virtex-II Pro development board may be found in the directory:

/CMC/Xilinx_Boards/XUP_V2PRO_BOARD.

This directory contains Postscript files for HardWare Reference Manual as well as complete schematic diagrams for the board.

PART V: Command Line Interface

This section explains how to use the Precision RTL synthesis, Xilinx ISE, and Xilinx Impact software tools from the Linux/UNIX command line instead of running the software tools through use of the various graphical user interfaces (GUIs) provided by the software tools. It is very useful to be able to run the tools from the command line for the following reasons:

• ease of use - the same basic steps are performed in the design flow, all that changes is the VHDL source code and perhaps some .ucf files (Xilinx user constraints file) and perhaps the target FPGA device. The use of UNIX scripts allows for rapid modification of existing scripts so that a new design may be implemented without having to redo the entire setup procedure with the various GUIs.

• scripts execute much faster - this is useful for large designs which may require significant processing time to complete.

• scripts may be executed in the background with the UNIX nohup command - long synthesis compiles may be run on a fast server without requiring any intervention from the user; background processes continue to execute even if you logout from the system which you initiated them on.

This section that the user is familiar with basic UNIX shell scripting. If you lack such experience, refer to any UNIX guide or textbook.

I. Running Precision RTL from the Command Line

Prior to invoking Precision in command line mode, it is necessary to source the fpga_advantage.env file to setup up the Linux environment. A typical command line to setup the environment is (make sure you have ssh into a Linux system) :

ted@focus FPGA_ADV 12:17pm > source /CMC/ENVIRONMENT/fpga_advantage_linux.env

The Precision RTL tool can now be invoked in non-GUI mode by using the command

```
precision -shell
```

In this mode, you can enter commands in an interactive manner. For example,

```
ted@brownsugar FPGA_ADV 12:19pm >precision -shell
precision: WARNING: Executing on unsupported platform: SunOS 5.9
precision: Setting MGC_HOME to /nfs/software/cmc/tools/MentorB.4/fa_71/Preci-
sion/Mgc_home ...
//  Precision Synthesis 2005a.69 (Production Release) Fri Jul 15 00:30:14 PDT
2005
//
//  Copyright (c) Mentor Graphics Corporation, 1996-2005, All Rights Reserved.
//              Portions copyright 1991-2004 Compuware Corporation
```

```
//                         UNPUBLISHED, LICENSED SOFTWARE.
//              CONFIDENTIAL AND PROPRIETARY INFORMATION WHICH IS THE
//            PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS
//
//  Running on SunOS ted@brownsugar.ece.concordia.ca Generic_118558-34 5.9
sun4u
//
//  Start time Thu Mar  8 12:22:27 2007
# -------------------------------------------------

# Logging session transcript to file "/nfs/home/t/ted/SYNOPSYS_2000/FPGA_ADV/
precision.log"
Precision{1}:
Precision{1}:
```

The tool displays some messages then issues the prompt `Precision{1}:`
You may enter Precision RTL commands at this prompt. A very useful command is the `help` command:

```
Precision{2}: help
#   "activate_impl" -- "activate the specified implementation"
#   "add_input_file" -- "Adds a file(s) to the input files list"
#   "add_macro_file" -- "Adds macro(s) file, .mdb, to the macro files list"
#   "add_placement_file" -- "Adds a physical database, pdb/fdb pair, to the
list of physical databases"
#   "alias" -- "define alternative command for a (set of) command(s)"
#   "all_clocks" -- "list all clocks"
#   "all_inouts" -- "list all the inout bidir ports"
#   "all_inputs" -- "list all the input ports"
#   "all_outputs" -- "list all the output ports"
```

The help command lists all the available commands (only a small portion of the total number of commands are listed above).

Information concerning a particular command may be obtained using `help command_name`:

```
Precision{3}: help add_input_file
#   "add_input_file" -- "Adds a file(s) to the input files list"
# usage : "add_input_file" File name(s) to be added to the list of input files
#    [-format <string>]      -- input format : vhdl|ver-
ilog|edif|syn|lib|tcl|xnf|xdb|sdf. Default will automatically detect the for-
mat.
#    [-work <string>]        -- specify library where design should be stored.
Default = work
#    [-exclude]              -- Exclude this file from the Compile phase.
#    [-reset]               -- Reset the existing list before adding the spec-
ified file(s).
#    |[-insert_before <integer>]-- Add this file before file number N. If not
set append the file to the end of the list.
#    |[-insert_after <integer>]-- Add this file after file number N. If not set
append the file to the end of the list.
```

```
#    |[-replace]             -- Replace existing file in list with these new 
settings
#    [-search_path <list>]   -- Set the search path for included files.
#    [-compile_time <integer>]-- Set the compile timestamp.
```

A more general approach is to use the command line to specify a Tcl (Tool command language file) which consists of various commands to be executed. These commands typically set constraints and compile and synthesize the design. The following command line is used to specify which Tcl command file is to be used:

```
precision -shell -file do_file.tcl
```

In the above command, `do_file.tcl` is the name of a text file which contains the following:

```
# This is a comment line
# Ted Obuchowicz
# Feb. 21, 2007
# sample script file
# Precision Synthesis interprets the backslash (\)
# as a Tcl escape character
# so \\ means the command is continued on the next line


new_project -name Test_Script -folder    \\
/nfs/home/t/ted/SYNOPSYS_2000/FPGA_ADV   \\
-createimpl_name Test_Script_impl

add_input_file ../Code/3_bit_counter.vhd

setup_design -frequency=100
setup_design -manufacturer Xilinx -family {VIRTEX-II Pro}  \\
-part 2VP30ff896 -speed 7

compile
synthesize
save_impl
save_project
exit
```

The end result of executing the `precision -shell -file do_file.tcl` command is the generation of the EDIF netlist in the directory specified by the -createimpl_name which in this example is the directory called Test_Script_impl. If you wish to view the schematic of your synthesized circuit, you may invoke the GUI version of precision and open the appropriate .psp file. In this example, it is the Test_Script.psp file.
The next step is to run the Xilinx tools from the Solaris command line to generate the .bit and .ace files.

II. Running Xilinx ISE from the Command Line

Create a subdirectory in your Xilinx directory which will be used to hold files required as input to the Xilinx tools and to save any generated output files. For this example, the chosen directory name was called `my_scr`. Copy the .edf file generated during the running of the precision -shell command into this directory. It will also be necessary to create any .ucf file if you wish to map input/output ports to specific pins of the FPGA device on the development board. Create a text file in this directory containing the following lines (save it with an appropriate name such as `counter_3_bit_pretty.scr`):

```
#!/bin/csh
#  is the line continuation character

source /CMC/ENVIRONMENT/xilinx.env


ngdbuild -intstyle ise -dd "/nfs/home/t/ted/SYNOPSYS_2000/Xilinx/my_scr/_ngo"  \
        -uc counter_3_bit.ucf  -p xc2vp30-ff896-7 counter_3_bit.edf counter_3_bit.ngd

map -intstyle ise -p xc2vp30-ff896-7 -cm area -pr b -k 4 -c 100 -tx off \
    -o counter_3_bit_map.ncd counter_3_bit.ngd counter_3_bit.pcf

par -w -intstyle ise -ol std -t 1 counter_3_bit_map.ncd counter_3_bit.ncd counter_3_bit.pcf

trce -intstyle ise -e 3 -l 3 -s 7 -xml counter_3_bit.twx counter_3_bit.ncd \
     -o counter_3_bit.twr counter_3_bit.pcf

netgen -intstyle ise -s 7 -pcf counter_3_bit.pcf -rpw 100 -tpw 0 \
       -ar Structure -xon true -w -ofmt vhdl -sim counter_3_bit.ncd counter_3_bit_gate.vhd

bitgen -intstyle ise -f counter_3_bit.ut counter_3_bit.ncd
```

Note that in the line:

```
ngdbuild -intstyle ise -dd "/nfs/home/t/ted/SYNOPSYS_2000/Xilinx/my_scr/_ngo"
```

you should change the specified path name to reflect your actual path to where you have created your Xilinx/my_scr directory (instead of /nfs/home/t/ted/SYNOPSYS_2000/).

It is necessary to add execute permission to this file, since it will be run as a Unix shell script. This is done with the chmod command:

```
chmod u+x  counter_3_bit_pretty.scr
```

if you perform a UNIX listing of this file, you will now see x in the permission triplet:

```
ted@brownsugar my_scr 12:20pm >ls -al counter_3_bit_pretty.scr
-rwx------ 1 ted ted 833 May  2 11:46 counter_3_bit_pretty.scr
```

It is also necessary to have a file called counter_3_bit.ut in your directory with the following contents:

```
-w
-g DebugBitstream:No
-g Binary:no
-g CRC:Enable
-g ConfigRate:4
-g CclkPin:PullUp
-g M0Pin:PullUp
-g M1Pin:PullUp
-g M2Pin:PullUp
-g ProgPin:PullUp
-g DonePin:PullUp
-g TckPin:PullUp
-g TdiPin:PullUp
-g TdoPin:PullUp
-g TmsPin:PullUp
-g UnusedPin:PullDown
-g UserID:0xFFFFFFFF
-g DCIUpdateMode:AsRequired
-g StartUpClk:JtagClk
-g DONE_cycle:4
-g GTS_cycle:5
-g GWE_cycle:6
-g LCK_cycle:NoWait
-g Security:None
-g DonePipe:No
-g DriveDone:No
-g Encrypt:No
```

This file is needed by the bitgen command (its use is explained in a later section).

You may now run the shell script. The tee command is useful for logging the screen output to a text file which may be reviewed once the script has finished its execution:

```
ted@brownsugar my_scr 12:21pm >counter_3_bit_pretty.scr | tee script.logfile
```

The script will start to run and produce messages as it proceeds. Let us know examine the various commands contained in the script:

```
ngdbuild -intstyle ise -dd "/nfs/home/t/ted/SYNOPSYS_2000/Xilinx/my_scr/_ngo"  \
        -uc counter_3_bit.ucf  -p xc2vp30-ff896-7 counter_3_bit.edf counter_3_bit.ngd
```

The ngdbuild command translates and merges the various source files of a design into a single "NGD" design database, this is a binary format used by the Xilinx tools. The various command line options are:

`-dd output_dir:` Directory to place intermediate .ngo files

`-intstyle ise|xflow|silent:` Indicate contextual information when invoking Xilinx applications

`-uc ucf_file:` Use specified "User Constraint File".

`-p partname:` Use specified part type to implement the design

The two input files to ngdbuild are the `counter_3_bit.ucf` and the `counter_3_bit.edf` file produced by precision. The end result of executing ngdbuild is the output file counter_3_bit.ngd.

The map command:

```
map -intstyle ise -p xc2vp30-ff896-7 -cm area -pr b -k 4 -c 100 -tx off \
    -o counter_3_bit_map.ncd counter_3_bit.ngd counter_3_bit.pcf
```

is used to “map the logic gates of the user’s design (previously written to an NGD file by NGDBUILD) into the CLBs and IOBs of the physical device, and writes out this physical design to an NCD file”. The input to the map command is the counter_3_bit.ngd file and the outputs produced are the counter_3_bit_map.ncd and counter_3_bit.pcf.

The par command:

```
par -w -intstyle ise -ol std -t 1 counter_3_bit_map.ncd counter_3_bit.ncd counter_3_bit.pcf
```

is used to “places and route a design’s logic components (mapped physical logic cells) contained within a NCD file based on the layout and timing requirements specified within the Physical Constraints File (PCF)”. The input files are counter_3_bit.ncd and counter_3_bit.pcf, the output file produced by map is the counter_3_bit_map.ncd. The command line options are:

-w = Overwrite. Allows overwrite of an existing file

-ol = Overall effort level. high is maximum effort, Default: std (standard)

-t = Placer cost table entry. Start at this entry., Default: 1.

The trce command is used to “Creates a Timing Report file (TWR) derived from static timing analysis of the Physical Design file (NCD). The analysis is typically based on constraints included in the optional Physical Constraints file (PCF):

```
trce -intstyle ise -e 3 -l 3 -s 7 -xml counter_3_bit.twx counter_3_bit.ncd  \
    -o counter_3_bit.twr counter_3_bit.pcf
```

The two input files to trce are counter_3_bit.ncd and counter_3_bit.pcf. The outputs are specified with the -xml and -o options:

```
-o <report[.twr]>  ... optional report output file (default design.twr)
-xml <xmlfile>     ... optional XML report output file (can be any extension;
                       default is .twx)
```

The netgen command is used to create the gate-level VHDL simulation file:

```
netgen -intstyle ise -s 7 -pcf counter_3_bit.pcf -rpw 100 -tpw 0 \
      -ar Structure -xon true -w -ofmt vhdl -sim counter_3_bit.ncd counter_3_bit_gate.vhd
```

The command “extracts design data from NCD, NGA, NGC, NGD or NGO input file and generates a VHDL netlist compatible with supported simulation tool.” The use of gate-level simulation is exlained in another section.

The last command bitgen is used to “create the configuration (BIT) file based on the contents of a physical implementation file (NCD). The BIT file defines the behavior of the programmed FPGA.” :

```
bitgen -intstyle ise -f counter_3_bit.ut counter_3_bit.ncd
```

The -f option is used to specify a command file which is used by the bitgen command. This command file specifies the use of the JtagClk as the StartUp Clock. It is necessary that you have a file called counter_3_bit.ut in your directory containing the following:

```
-w
-g DebugBitstream:No
-g Binary:no
-g CRC:Enable
-g ConfigRate:4
-g CclkPin:PullUp
-g M0Pin:PullUp
-g M1Pin:PullUp
-g M2Pin:PullUp
-g ProgPin:PullUp
```

-g DonePin:PullUp
-g TckPin:PullUp
-g TdiPin:PullUp
-g TdoPin:PullUp
-g TmsPin:PullUp
-g UnusedPin:PullDown
-g UserID:0xFFFFFFFF
-g DCIUpdateMode:AsRequired
-g StartUpClk:JtagClk
-g DONE_cycle:4
-g GTS_cycle:5
-g GWE_cycle:6
-g LCK_cycle:NoWait
-g Security:None
-g DonePipe:No
-g DriveDone:No
-g Encrypt:No

III. Running the Xilinx Impact tool from the Command Line:

The command to run the Xilinx Impact tool to generate the SystemAce .ace file is:

```
impact -batch impact.batch_file
```

Make sure that you have source the xilinx.env file prior to entering this on the command line. The impact.batch_file is a text file containing a list of commands to the impact program. This file is exactly the same as the `_impact.cmd` file created when running the Impact program in the GUI mode. You may simply edit this file to change the relevant paths to the input .bit file and other files. The contents of the impact.batch_file used in this example is:

setPreference -pref UserLevel:NOVICE
setPreference -pref MessageLevel:DETAILED
setPreference -pref ConcurrentMode:FALSE
setPreference -pref UseHighz:FALSE
setPreference -pref ConfigOnFailure:STOP
setPreference -pref StartupCLock:AUTO_CORRECTION
setPreference -pref AutoSignature:FALSE
setPreference -pref KeepSVF:FALSE
setPreference -pref svfUseTime:FALSE
setPreference -pref UserLevel:NOVICE
setPreference -pref MessageLevel:DETAILED
setPreference -pref ConcurrentMode:FALSE
setPreference -pref UseHighz:FALSE
setPreference -pref ConfigOnFailure:STOP

setPreference -pref StartupCLock:AUTO_CORRECTION
setPreference -pref AutoSignature:FALSE
setPreference -pref KeepSVF:FALSE
setPreference -pref svfUseTime:FALSE
setMode -cf
setMode -cf
setAttribute -configdevice -attr path -value "/nfs/home/t/ted/SYNOPSYS_2000/Xilinx/my_scr"
setMode -cf
setAttribute -configdevice -attr size -value "134217728"
setAttribute -configdevice -attr reseveSize -value "0"
setAttribute -configdevice -attr name -value "XCCACE128-I"
addCollection -name "imp_scri"
addDesign -version 0 -name "rev0"
addDeviceChain -index 0
setCurrentDesign -version 0
addDevice -position 1 -file "/nfs/home/t/ted/SYNOPSYS_2000/Xilinx/my_scr/counter_3_bit.bit"
setAttribute -configdevice -attr path -value "/nfs/home/t/ted/SYNOPSYS_2000/Xilinx/my_scr"
setMode -cf
generate -active imp_scri
setMode -pff
setMode -sm
setMode -cf
setMode -cf
setMode -pff
setMode -sm
setMode -cf
setMode -bs
setMode -ss
setMode -sm
setMode -bsfile
setMode -dtconfig
setMode -cf
setMode -mpm
setMode -pff
setMode -cf
setMode -cf
quit

File names which need to be modified are indicated in bold font. The .ace file is created and saved in the imp_scr/rev0 directory (or whichever directory is specified by the addCollection -name " command contained in the file).

REFERENCES

1. Modelsim SE Reference Manual, v6.6d, p. 306.

2. Precision RTL Synthesis User's Manual, Mentor Graphics, p. 1-1.

3. http://www.digilentinc.com

APPENDIX 1: SWITCH AND LED OPERATION