INTRODUCTION

This tutorial guide is an introduction to digital logic synthesis using the Synopsys and Xilinx tools. You should have working knowledge of the UNIX 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 using the tools rather than learning how to write VHDL code. There are many fine books dealing with VHDL. There are not many books concerning the use of synthesis tools. This tutorial attempts to bridge the gap between the novice users's 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 tool. 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 four parts. Part I deals with VHDL simulation using the Synopsys VHDL System simulator (VSS). Part II focuses on logic synthesis; the Synopsys Design Compiler and Design Analyzer being the tools of choice. Design Compiler shell scripts have been used rather than the alternative Design Analyzer Graphical User Interface method of synthesizing a design. This conscious decision was based upon the convenience of shell scripts: they may be executed from the command line (obviating the need for a graphics workstation terminal) and they may readily modified for other designs. Part III concerns itself with implementation using the Xilinx Design Manager. 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. The last part (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.

The designs in this tutorial were synthesized using Synopsys's Design Compiler Version 3.4b and Xilinx's Alliance M1.3 software.

PART I : VHDL Simulation using SYNOPSYS

This section explains the use of the Synopsys 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. While more information can be generated and absorbed by the user in graphical mode, the use of the command line version allows remotely accessing and using the tools via a modem connection.

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

Prior to running the Synopsys tools, it is necessary to set up your UNIX computer account. Perform the following from your UNIX prompt. In the following, the % symbol refers to the UNIX prompt, your prompt may be different.

Step 1:

% source /home/ted/ENVIRONMENT/synopsys.env

Alternatively, once may copy the file /home/ted/ENVIRONMENT/synopsys.env to one's home directory and source it from there:

% cd
% cp /home/ted/ENVIRONMENT/synopsys.env .
% source synopsys.env

Step 2:

The Synopsys tools requires several setup files in order to function properly. In order to perform VHDL simulation, a file named .synopsys_vss.setup is needed. There are three locations for this file:

(i) the default system setup file found in the Synopsys installation directory which in our current setup is /CMC/tools/synopsys/admin/setup/.synopsys_vss.setup

(ii) the user's home directory

(iii) the current working directory.

The Synopsys tools read the required setup files in the above specified order. Each successive reading will override any previous settings found in a previously read file. For our purposes, it is simplest to create a .synopsys_vss.setup file in a working directory from which we will invoke the simulation tools from. We will first create a directory called Synopys, 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, the .synopsys_vss.setup file, and a directory called Work (which will be used to hold intermediate files created by the simulation tools). Figure 1 illustrates the direc-

tory hierarchy which will be created.

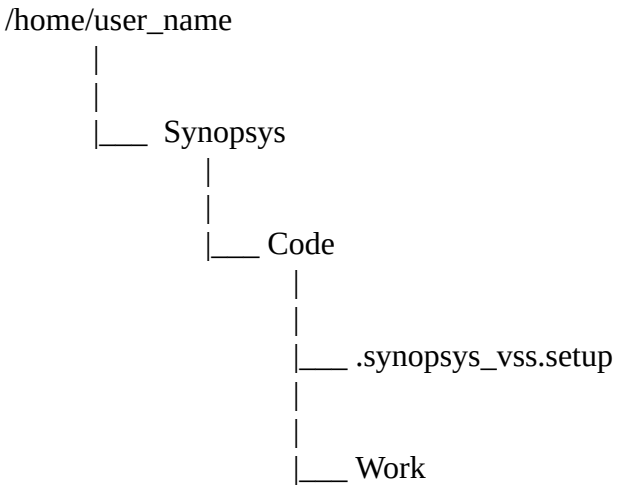

Figure 1: Directory hierarchy for VHDL Simulation using Synopsys.

Issue the following commands from the UNIX prompt:

```
% cd
% mkdir Synopsys
% cd Synopsys
% mkdir Code
% cd Code
% mkdir Work
% cp /home/ted/SYNOPSYS/Code/.synopsys_vss.setup .
```

This sequence of commands will create the Synopsys directory, the Code subdirectory, the Work subdirectory and will copy the required setup file to the Code subdirectory. Examine the contents of the .synopsys_vss.setup file. It contains three lines of text:

```
WORK > DEFAULT
DEFAULT: ./Work
TIMEBASE = NS
```

The first two lines tell the Synopsys tools that the logical library named WORK should be mapped to the physical UNIX directory called ./Work. The . is UNIX shorthand notation for the current directory. The last line sets the default timebase to nanoseconds. You need not concern yourself further with this file.

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

ulation.

II. Performing VHDL simulation using Synopsys

This section will illustrate the use of the Synopsys 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 Synopsys/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 UNIX text editor (vi, emacs, xedit, etc) to create and save this file. The next step is to "analyze" 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 analysis of VHDL code, any syntactical errors will be reported.

There are two Synopsys tools used to analyze VHDL code. The first is called **vhdlan**, and is the command line version. This means that vhdlan can be invoked through a modem connection.

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

% vhdlan and2.vhd

A small message giving the version number of the tool will be displayed and you will be returned to the UNIX prompt if your code contains no syntax code. A typical session with **vhdlan** is as given below:

ted@dea Code 3:57pm >vhdlan and2.vhd
Synopsys 1076 VHDL Analyzer Version 3.4b

```
        authorized in a  license agreement  controlling such use
        and disclosure.


ted@dea Code 4:06pm >
```

If there are syntactical errors in the source code, vhdlan 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@dea Code 4:11pm >vhdlan and2.vhd
Synopsys 1076 VHDL Analyzer Version 3.4b

            Copyright (c) 1990-1995 by Synopsys, Inc.
                    ALL RIGHTS RESERVED
        This program is proprietary and confidential information
        of  Synopsys, Inc. and may be used and disclosed only as
        authorized in a  license agreement  controlling such use
        and disclosure.


port( in_1, in_2: in bitt;
                  ^
**Error: vhdlan,575 and2.vhd(2):
    BITT is not declared.
“and2.vhd”: errors: 1; warnings: 0.
ted@dea Code 4:11pm >
```

Vhdlan reports the line number which the error(s) occur(s) enclosed in parentheses following the file name. In this case, there is an error in line 2 of the file and2.vhd.

(3) Analysis may also be performed using **gvan**, which is the graphical-user-interface equivalent of **vhdlan**. In order to use gvan, one must be sititng in front of a graphics workstation.

```
% gvan and2.vhd
```

As in the case, of vhdlan, gvan will perform its work and return to the UNIX prompt in the case of no errors in the input file. If you change into the Work subdirectory, you will see that two files were created as a result of performing analysis. In this example the two files are named AND2_GATE.mra and AND2_GATE.sim. Had we not told Synopsys (through the .synopsys_vss.setup file) to store these files in the Work subdirectory, the tool would have created them in the directory from which the command was invoked from.

In the case where the source file contains errors, an X-window will be opened on your display (see Figure 2). Any errors contained in the source code will be displayed in the central portion of

the window. One may choose to correct the error by selecting with the left mouse button the Edit button found at the bottom of the window. This will open a new window with the vi editor invoked upon the file in question. Alternatively, one may exit gvan (by selecting Cancel) and edit the offending line using any available text editor.

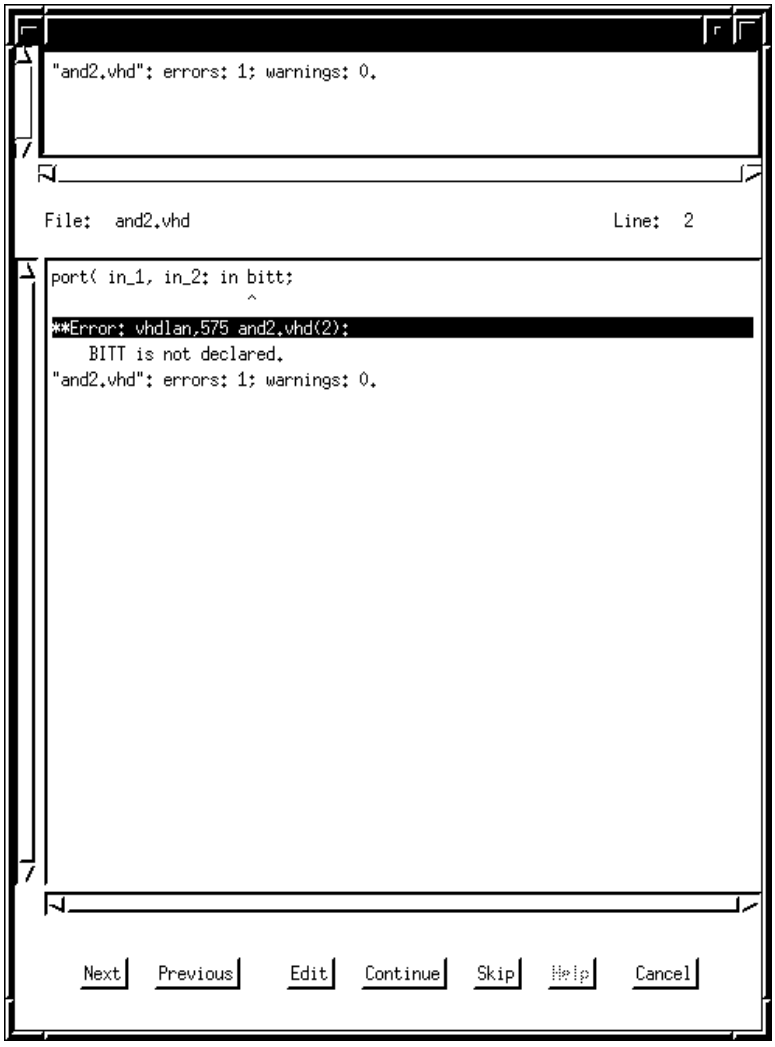

Figure 2: Gvan window containing error messages.

(4) The next step is to simulate the VHDL model. This is done with the **vhdldbx** command. This command will invoke the graphical VHDL debugger tool. From the UNIX prompt enter:

% vhdldbx &

The ampersand character (&) will cause the system to return you to your UNIX prompt after the **vhdldbx** window appears (see Figure 3).

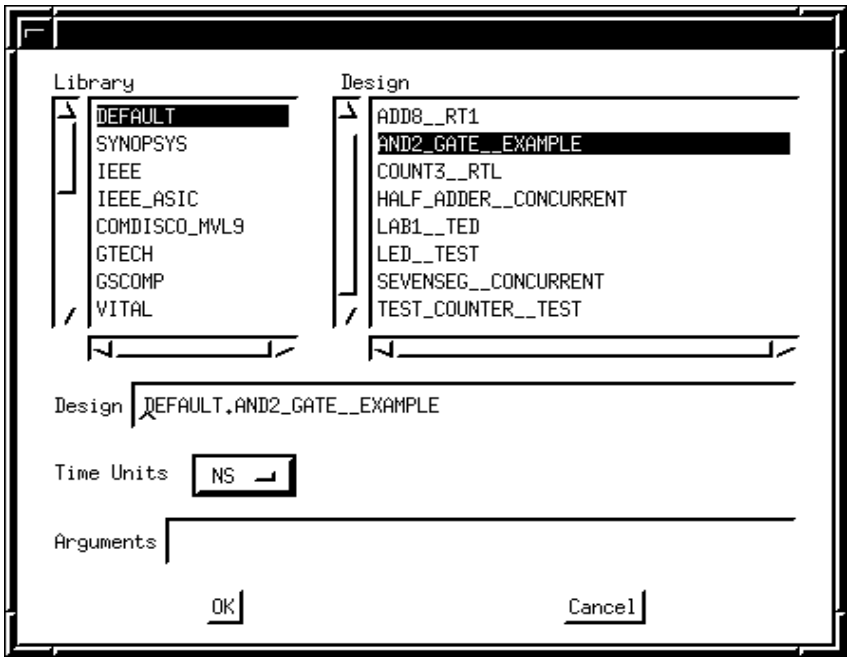

Figure 3: Vhdldbx window.

Take a moment to examine this window. The Library scrollbox contains a list of libraries. The default library is set to DEFAULT. If you recall, this library was mapped to the logical library called WORK which was mapped to the physical directory .Work. This was accomplished with the .synopsys_vss.setup file. When vhdlbx is invoked, the tool searches the selected library and will list in the design scrollbox the names of any entity-architecture pairs using the format entityname__architecturename. In my Work library, I had many compiled entity-architecture pairs, hence the long list. You should only see one listed: AND2_GATE__EXAMPLE. Select this design with the left mouse button.

The Time Units button is used to select the time unit used during the running of the simulation. Its value may be altered by selecting a different value from the list which will appear when this button is selected. To invoke the simulator, select the OK button. The Synopsys VHDL Debugger (Vhdldbx) window will appear as shown in Figure 4. This window contains the source code of the current working region being simulated (more on the concept of current working region in a later section of this tutorial). At the bottom of the window is a command window where simulator commands may be entered manually. The # symbol designates the simulator prompt. In the central portion of the window, the current value of the simulation time is listed.

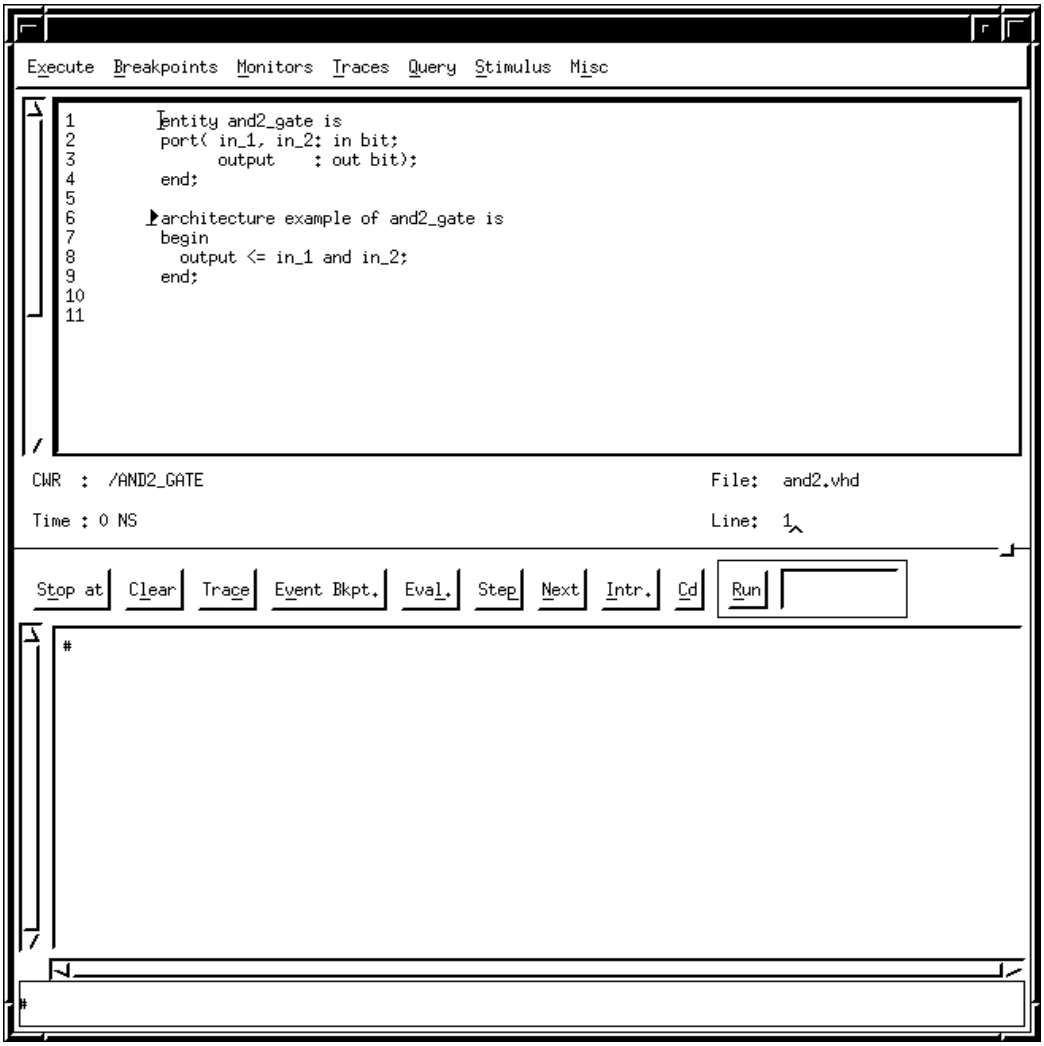

Figure 4: Synopsys VHDL Debugger (Vhdldbx) window.

(5) One can manually assign values to signals using the **trace** and **assign** commands. The trace command will cause the signal which is to be traced to be listed in a waveform window. The assign command is used to set a signal to a certain value. For example, we wish to have the values of the three signals in_1, in_2, and output appear in the waveform window. To do so we would issue the following commands from the command window of the Debugger window (see Figure 5):

trace in_1
trace in_2
trace output

Alternatively, one may list all signals to be traced on a single line separated by one or more blank spaces:

trace in_1 in_2 output

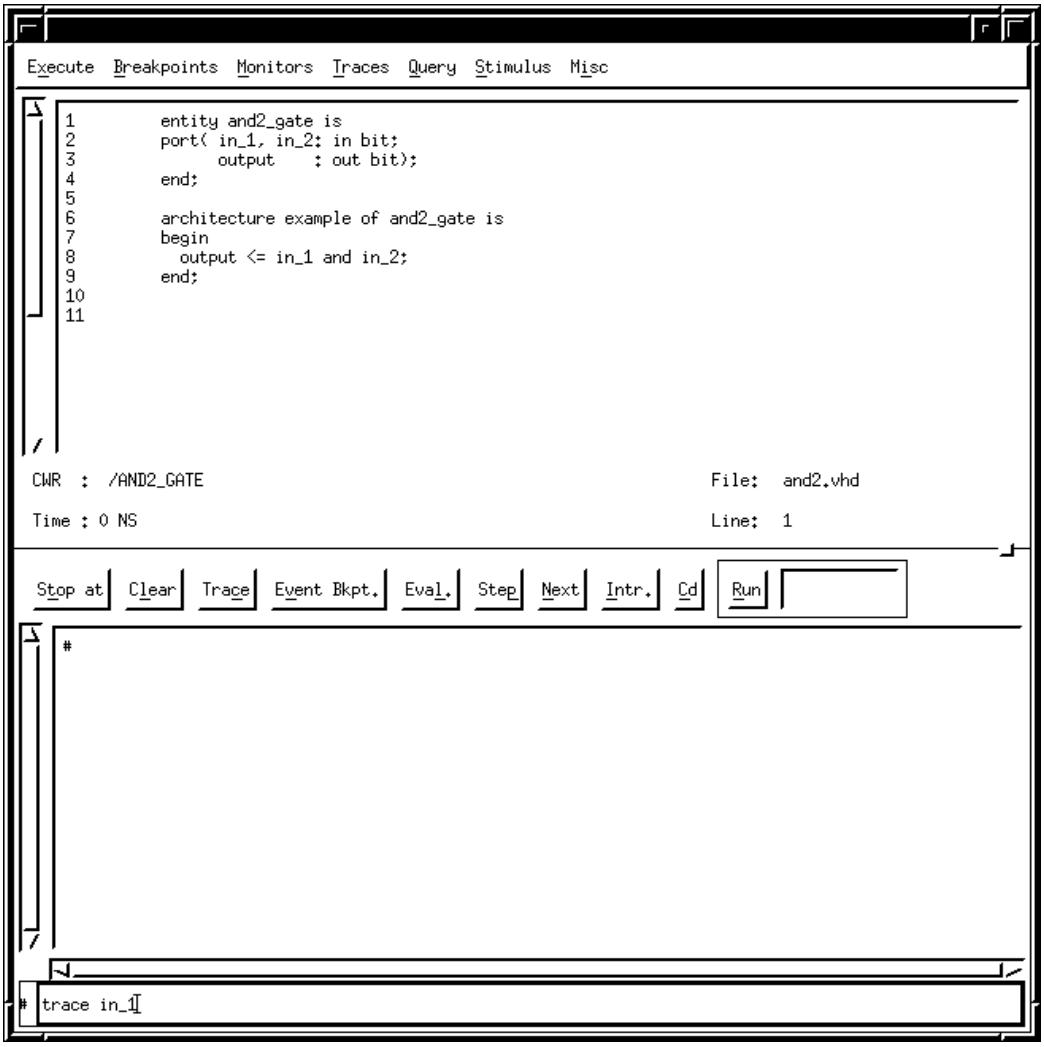

Figure 5: Issueing a trace command from the command window in Vhdldbx.

After entering this command, the system may report that it is builing a cache for the fonts directory. Be patient this may take some time on slow machines. This cache is built only once, future simulations will refer to this fonts cache directory. You will see listed in the Debugger window the following messages:

trace in_1
stlpatch: Can’t write to temporary
stlpatch: Error 0
stlpatch: Can’t write to temporary
stlpatch: Error 0
#

Ignore the stlpatch: messages. After a while, the Synopsys Waveform Viewer window will appear as shown in Figure 6.

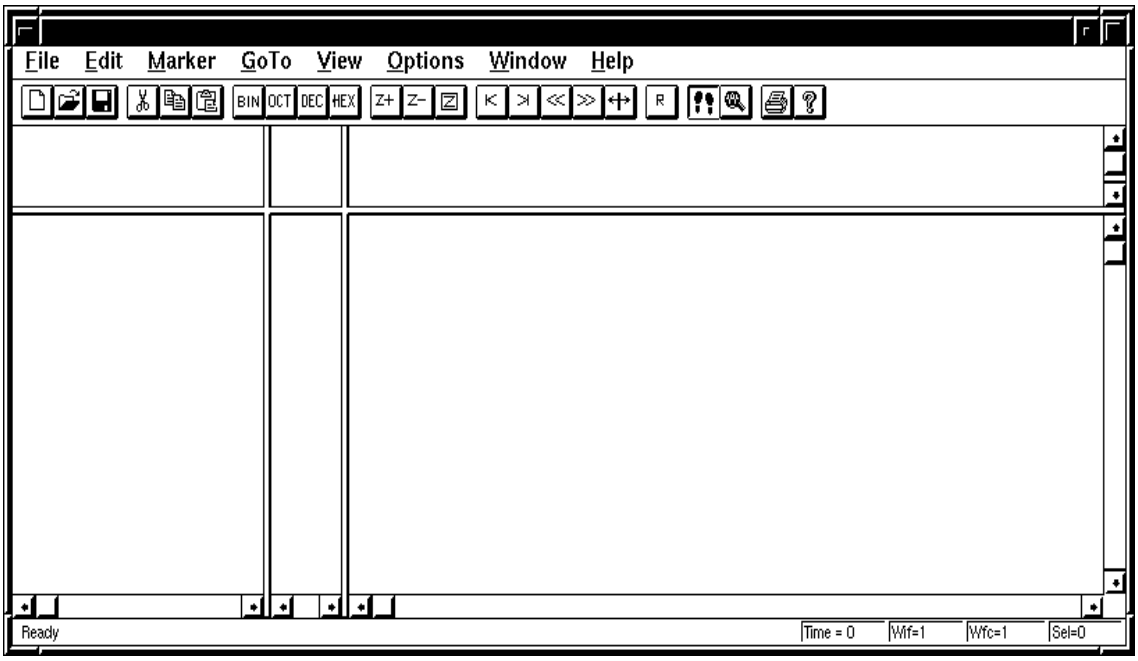

Figure 6: The Synopsys Waveform Viewer window after a trace in_1 command.

(6) After tracing the desired signals, one may assign values to signals with the **assign** command. For example to set the value of the signal in_1 to value ‘1’ use the following command:

assign ‘1’ in_1

Similarly, we can assign the value ‘1’ to input signal in_2 with the command:

assign ‘1’ in_2

(7) Run the simulation for a period of 2 ns. Issue the command:

run 2

The waveforms for the selected signals will be updated in the waveform viewer. To see the complete waveform, select View -> Full Fit from the Waveform window top menu bar. You can now assign new values to the inputs, run the simulator for another time period and view the new results. Experiment with this method of assigning values and viewing results. Figure 7 shows the results of the simulation run. You may print the simulation results to a Postscript file by selecting File -> Print . Select the File choice circle in the Print To section and specify an appropriate file-

name. Select Ok to print the file. Refer to the Appendix section for a printout of a sample simulation session of this VHDL model of a simple two-input and gate. This concludes this introductory example.

Example 2: Simulating a two-input and gate (using the command line interface)

This example uses the same and2.vhd file as the previous one; however the use of the command line **vhdlsim** tool is illustrated. This method can be used when access to a graphics terminal is not possible.

(1) Analyze the file using vhdlan:

%vhdlan and2.vhd

(2) To invoke the Synopsys VHDL System Simulator (VSS) in command line mode use the command **vhdlsim** entity_name, where entity_name refers to the name of the entity in the VHDL source file, for example and2_gate. When a design is simulated with vhdlsim, you control and monitor the simulation by entering special **Simulation Control Language** commands at the simulator prompt which is represented by the # symbol. This example will illustrate the use of a few basic Simulation Control Language commands.

```
ted@dea Code 4:48pm >vhdlsim and2_gate
Synopsys 1076 VHDL Simulator Version 3.4b

        Copyright (c) 1990-1995 by Synopsys, Inc.
                ALL RIGHTS RESERVED
    This program is proprietary and confidential information
    of  Synopsys, Inc. and may be used and disclosed only as
    authorized in a  license agreement  controlling such use
    and disclosure.


#  assign ‘1’ /and2_gate/in_1
# assign ‘1’ /and2_gate/in_2
# run 2
2 NS
# fprint “%r\n” /and2_gate/output
‘1’
# assign ‘0’ /and2_gate/in_1
# run 2
4 NS
# fprint “%r\n” /and2_gate/output
‘0’
#quit
ted@dea Code 4:53pm >
```

The assign command is the same as that used in graphical mode. It is used to give a signal a value. Note how the signal names have been written: /and2_gate/in_1, /and2_gate/in_2, /and2_gate/output. In general, Synopsys uses a naming convention very similar that that employed by the UNIX file system. Names of VHDL objects are written in a hierarchical fashion beginning with the top level entity. In this example, the name of the top level entity being simulated is and2_gate, hence the full names of the signals in_1, in_2, and output declared within this entity are /and2_gate/in_1,/and2_gate/in_2, and /and2_gate/output.

The fprint command is used to output the value of an object. The format of this command is:

fprint "%format_specifier" argument

where format_specifier is one of:

b : the argument to fprintfis a single dimension vector of type bit, output given in binary
o : the argument to fprint is a single dimension vector of type bit, output given in octal
x : the argument to fprint is a single dimension vector of type bit,, output given in hexadecimal:
r: the argument to fprint is a VHDL expression. Fprint will print its value in a "reasonable" form
c: the argument to fprint is a character literal
s: the argument to fprint is a string
d: the argument to fprint is an integer type, output will be given in decimal format
e,f,g: the argument to fprintf is a floating-point type.

Within the format_string, you may represent certain non-graphic characters and the backslash \ using the following escape sequences.

\n LF, newline
\t HT, horizontal tab
\\ \, backslash

Other useful Simulation Control Language commands are:

step: used to single step through the VHDL source code line by line
next: similar to step except that it does not enter into functions or procedures
eval: used to print the value of any VHDL expression, handy for signals and variables
restart: used to restart the simulation and set the current value of simulation time to 0
interrupt: will interupt the simulation and return control to the user. Useful when your simulation goes awry. It sometimes takes a few seconds to return control.
help: used to obtain more information on commands

Example 3: Using Command Files to control simulation.

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. The format for this file is as follows:

comm file_name
command_1
command_2
...
...
command_n
end comm

This example illustrates the use of command 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 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 analyze these three files in the following order:

% gvan tedand.vhd
% gvan tedor.vhd
% gvan tedcircuit.vhd

(5) Create a file called tedcircuitstimulus which contains the following:

```
comm tedtester

trace /tedcircuit/a
```

```
trace /tedcircuit/b
trace /tedcircuit/c
trace /tedcircuit/d
trace /tedcircuit/s1
trace /tedcircuit/s2
trace /tedcircuit/e

assign ‘1’ /tedcircuit/a
assign ‘0’ /tedcircuit/b
assign ‘1’ /tedcircuit/c
assign ‘1’ /tedcircuit/d

run 10

assign ‘0’ /tedcircuit/a
assign ‘1’ /tedcircuit/b
assign ‘0’ /tedcircuit/c
assign ‘0’ /tedcircuit/d

run 10

assign ‘1’ /tedcircuit/a
assign ‘1’ /tedcircuit/b
assign ‘1’ /tedcircuit/c
assign ‘1’ /tedcircuit/d

run 10

end comm
```

(6) Invoke the vhdldbx debugger and select the TEDCIRCUIT__TED_ARCH design and select OK.

(7) From the VHDL Debugger window enter the following from the simulator prompt:

include tedcircuitstimulus
tedtester

(8) The simulator will open the Waveform Viewer window (since the command file included a trace command) and will execute the commands contained in the file tedcircuitstimulus. See the results of the simulation on the next page.

(9) At this point you may be wondering why it is necessary to specify signal names using the arcane format of *top_level_entity_name/signal_name*. The reason for this is that as the simulation proceeds, the simulation tools traverse the hierarchy. In the parlance of Synopys, The Current Working Region changes as the simulation proceeds. Observe the VHDL Debugger window after

issueing the commands listed in step 7 (see Figure 8). The source code window now lists the VHDL code for the entity-architecture of the ted_and gate. The Current Working Region is listed as /TEDCIRCUIT/U3/_P0. If one were to name signals without the full hierarchical path name, the signal would not be found once the simulation proceeded. For example, try to assign the top-level entity signal E a value of '0' from the simulator prompt:

assign '0' e

The simulator is not able to find this signal, since it is now longer in scope. The following error message is generated:

vhdlsim,575: E is not declared.

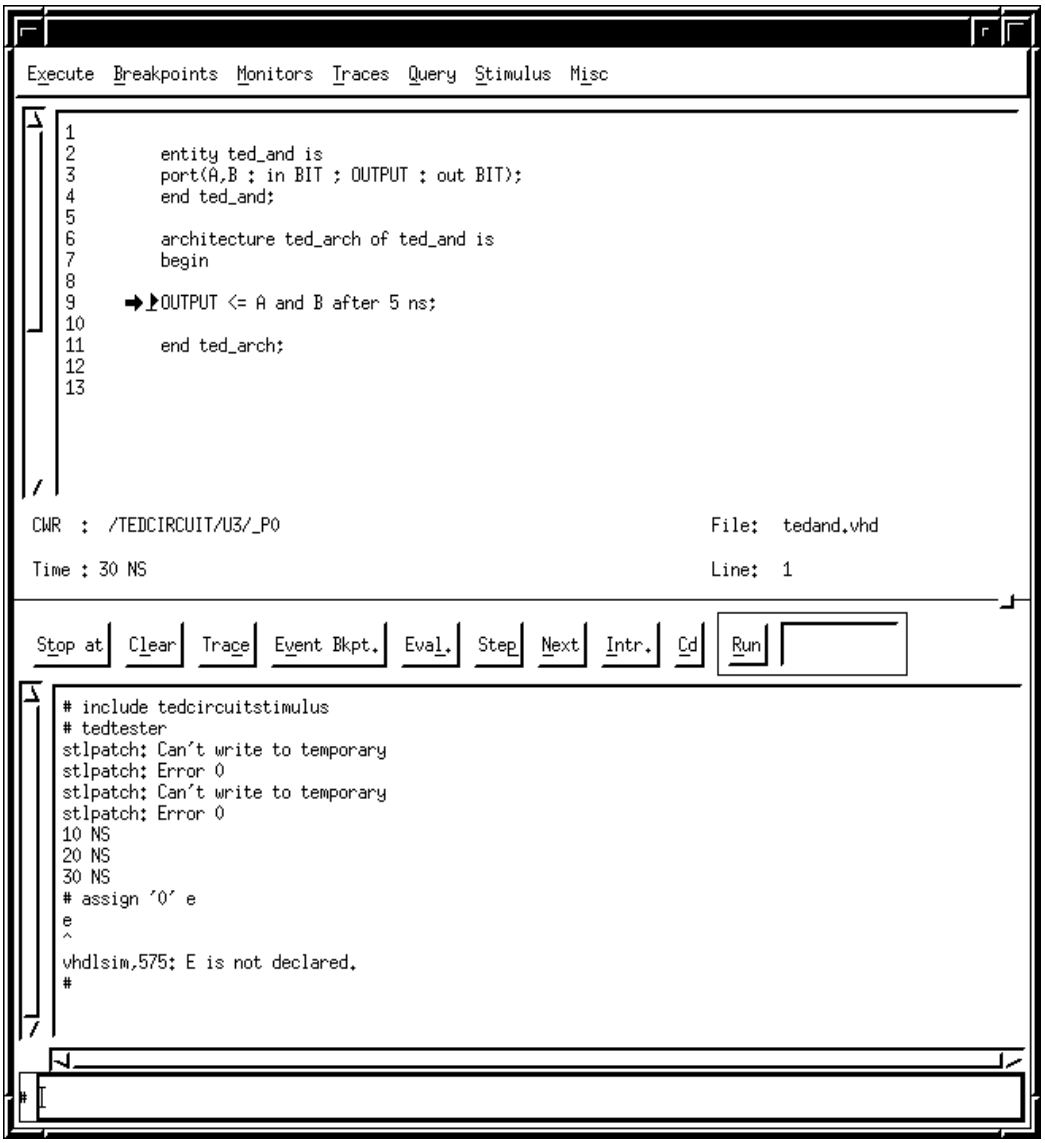

Figure 8: Descending to another level of hierarchy during simulation.

(10) We will now explore various techniques used to navigate a hierarchical design. Even the simplest circuits will exhibit some hierarchy, it is useful to move around through this hierarchy to examine the values of signals, variables, etc which are contained in different parts of the design. The vhdldbx debugger makes traversing a hierarchical design very simple, the commands are very similar to corresponding UNIX commands used to navigate throughout the file system (cd, pwd, ls).

Quit the current vhdldbx session, and restart it selecting the TEDCIRCUIT__TED_ARCH design. From the simulator prompt at the bottom of the debugger window type:

pwd

This command will report the which part of the circuit is currently in scope. In other words, pwd returns the value of the CWR (Current Working Region). In this example, vhdldbx will report / TEDCIRCUIT as the CWR.

Use the ls command to obtain a listing of the available circuit elements in this current working region:

ls
The following design elements are reported in the message window of the vhdldbx window:

```
A          D          U2          TED_OR
B          E          U3          S1
C          U1         TED_AND     S2
```

The TEDCIRCUIT top-level entity consists of the five signals A, B, C, D, and E, three component instantiation sttements labelled U1, U2, U3, two internal signals S1 and S2, and two components named TED_AND , TED_OR.

Use the ls -t command to obtain a list of the available components and their associated type:

ls -t

```
 A          IN PORT
  type = BIT
B          IN PORT
  type = BIT
C          IN PORT
  type = BIT
D          IN PORT
  type = BIT
E          OUT PORT
  type = BIT
U1         COMPONENT INSTANTIATION STATEMENT
U2         COMPONENT INSTANTIATION STATEMENT
```

```
U3            COMPONENT INSTANTIATION STATEMENT
TED_AND       COMPONENT
TED_OR        COMPONENT
S1            SIGNAL
  type = BIT
S2            SIGNAL
  type = BIT
```

Using ls with the -v option will print the value of all elements in the current working region (although not every design element will have a value).

```
# ls -v

A             '0'
B             '0'
C             '0'
D             '0'
E             '0'
U1            (no value)
U2            (no value)
U3            (no value)
TED_AND       (no value)
TED_OR        (no value)
S1            '0'
S2            '0'
```

We have not yet run the simulator, these values are the *initial* values as set by the simulator. By default elements of type "bit" are set to '0' during the initialization phase of simulation. Run the simulator (by including the tedcircuitstimulus file and running the tedtester command contained in it). Repeat the ls -v command to obtain the values at the end of the simulation.

Let's descend down into the hierarchy. Enter the command cd U1 from the simulator prompt. Enter ls to find out which elements are available. Since U1 is a component instnatiation statement which instantiates an instance of a TED_AND component, the available elements are the two input signals A, B, and the output signal OUTPUT.

Note that one may only cd into elements which have another level of hierarchy below. It makes no sense to cd into signals or component declarations. Attempting to do so will result in an error message being generated.

To return up to the previous level of hierarchy, use the cd .. command.

Example 4: This example will use the same three VHDL files as the previous example; another feature of the Synopsys tool will be explored. Specifically, we will use the Synopsys Design Analyzer to create a schematic diagram showing the interconnection of the various components and signals in our top-level entity tedcircuit. Prior to using the Design Analyzer, it is necessary to copy another Synopsys specific setup file to you Synopsys working directory. This file is the **.synopsys_dc.setup** file (Synopsys Design Compiler setup file). The use of this file will be explained in the Synthesis part of this tutorial. For the time being, issue the following commands to copy the file to your directory:

```
% cd
% cd Synopsys
% cp /home/ted/SYNOPSYS/.synopsys_dc.setup .
```

You may wish to change the line

```
designer = “Ted Obuchowicz” ;
```

in the .synopsys_dc.setup file to reflect your own name (such as “Keith Richards”). You may also want to copy the .synopsys_dc.setup file to your Code directory. This will allow you to invoke the Design Analyzer from your Code subdirectory and immediately obtain a listing of design, rather than having to move around your directory structure from within the Design Analyzer. This example assumes that there exists a copy of the .synopsys_dc.setup file in your Code subdirectory and your current working directory is the Code subdirectory. In Part II of this tutorial, we will invoke the Design Analyzer from the Synopsys directory, hence there should be a copy of the .synopsys_dc.setup file in this directory as well.

(1) Invoke the Synopsys Design Analyzer with the following command issued from the UNIX prompt (from your Code working directory):

```
% design_analyzer &
```

The Synopsys Design Analyzer will appear in a new window. See Figure 9 below.

Figure 9: Synopsys Design Analyzer window.

(2) From this window, Select File -> Analyze. A new Analyze File window will appear as shown in figure 10. From this window, select tedand.vhd from the file list and make surwe that the File Foramt is set to VHDL. Select OK.

(3) Repeat the steps given in (2) to analyze the files tedor.vhd, and tedcircuit.vhd. Note the order of analysis.

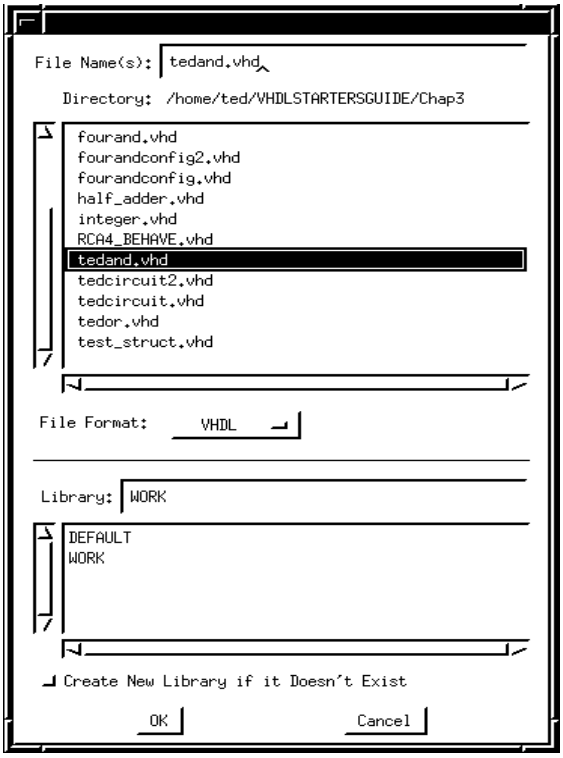

Figure 10: Analyze File window.

(4) From the Synopsys Design Analyzer window select File -> Elaborate. The Elaborate Design window will appear as illustrated in Figure 11. Select Default from the list of libraries listed in the top part of the form. A list of designs will then appear. Select tedcircuit(ted_arch) from as the Design and select OK.

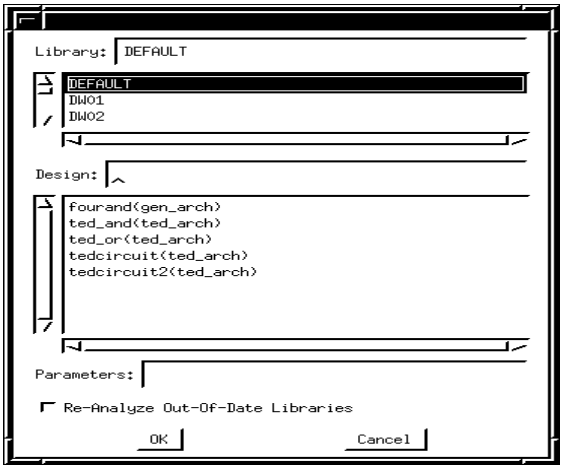

Figure 11: Elaborate Design Window.

(5) The central portion of the Design Analyzer window will now contain three yellow squares. These icons correspond to the entities which were analyzed. Use the left mouse button to click on the icon labelled tedcircuit.vhd. It will now be outlined with a dashed line. Select the down arrow button located on the bottom left hand side of the window; a new icon will appear in the central portion. There are four input ports labelled A, B, C, and D on the left hand side and a single output port labelled E. This represents the *symbol view* of the entity tedcircuit. (See Figure 12)

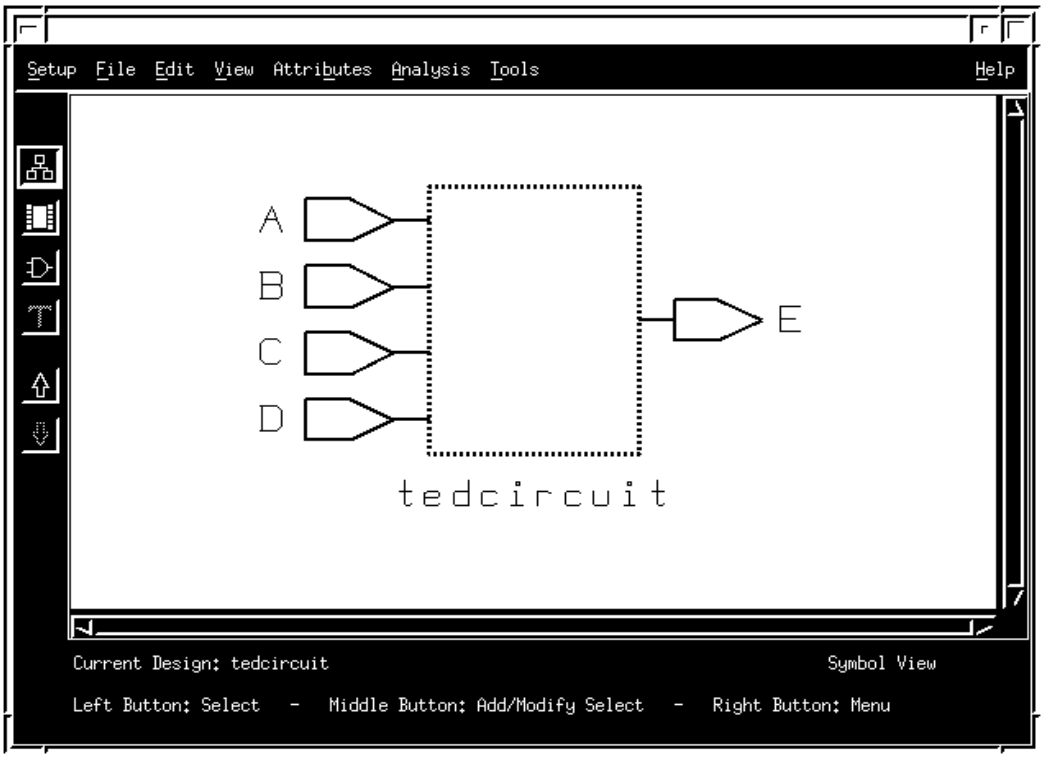

Figure 12: Design Analyzer window with symbol view of entity tedcircuit.

(6) Left click on the button labelled with the symbol for an AND gate located third from the top on the left hand side of the Design Analyzer window. The top-level icon is now replaced with a schematic representation of the VHDL code (see Figure 13). Use the View -> Zoom In to zoom in. Select one of the blue wires (which reprsent signals) with the left mouse button. The corresponding signal name listed in the field named Net located at the bottom left hand corner of the window.

Explore traversing the design hierarchy by selecting icon rectangle labelled ted_or and then selecting the down arrow. The symbol for the ted_or entity is shown with the two input ports and single output port. This is the bottom of the hierarchy; it is not possible to descend further. Note that the down arrow key is shown as not selectable in the window. To go up in the hierarchy, use the up arrow key, this will return you to the tedcircuit entity.

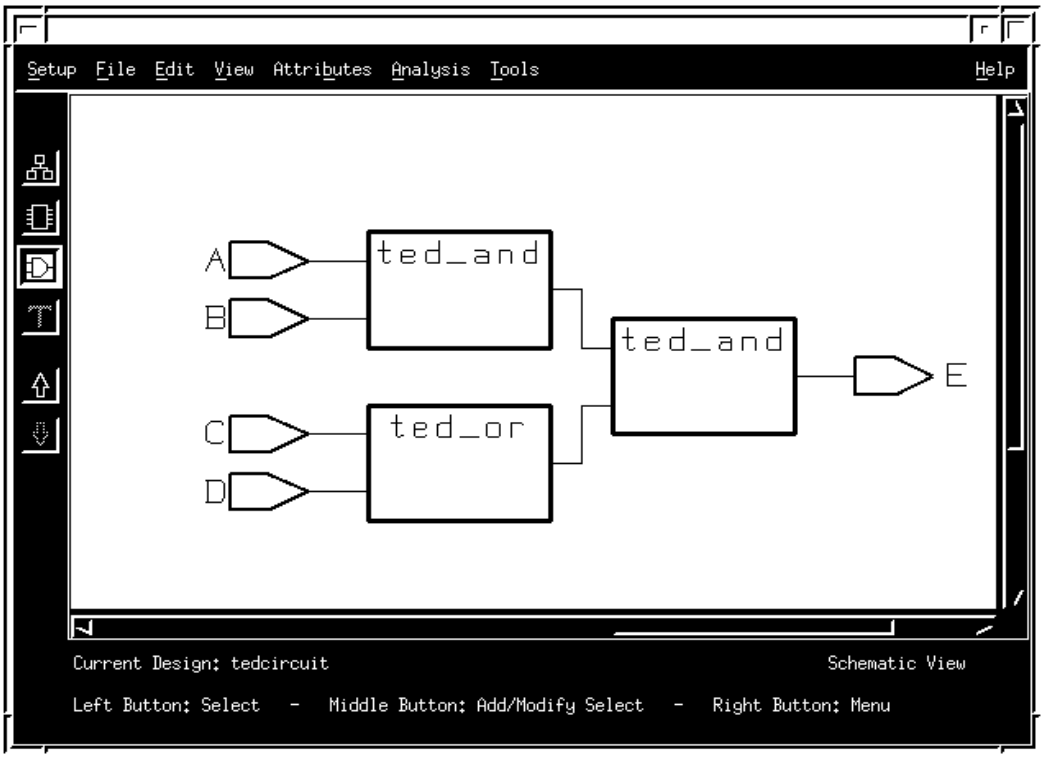

Figure 13: Schematic representation of the tedcircuit entity.

PART II : Logic Synthesis with SYNOPSYS

In this section we will use the Synopsys tools to perform logic synthesis. In synthesis, VHDL code will be translated into a 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 Alliance tool suite which will perform the translation from netlist file into a working design.

I. Setting up the technology libraries

At this point it is worthwhile to explain the purpose of the .synopsys_dc.setup file. The Synopys synthesis tools require some basic information concerning the target technology. In our case, the target technology is a Xilinx 4010e-3 Field Programmable Gate Array (FPGA). The -3 following the part number refers to the *speed grade* of the device. A smaller number indicates a higher speed device.

The .synopsys_dc.setup file which you have previously copied is already setup for a 4010e-3 device. If you wish to target a different device, the five lines specifying the link_library, target_library, define_design_lib, symbol_library, and synthetic_library must be replaced with those given by the **synlibs** command. Synlibs is part of the Xilinx tool suite; it displays the Synopsys link and target libraries that correspond to your choice of Xilinx part type/speed grade. In addition, synlibs will display the symbol library and synthetic library if available. You can then cut and paste these lines into your .synopsys_dc.setup file.

If you want to use the synlibs command, you must first set up your environment to run the Xilinx tools. Issue the following command from the UNIX prompt:

```
% source /home/ted/ENVIRONMENT/xilinx.env
```

Alternatively, you may copy this file to your directory and source it directly from there. Issue the synlibs 4010e-3 command to obtain the list of libraries:

```
ted@dea ~/SYNOPSYS 11:48am >synlibs 4010e-3
 libfam = 44 selection = 4010e-3 n = 38


link_library = {xprim_4010e-3.db xprim_4000e-3.db xgen_4000e.db xfpga_4000e-3.db
xio_4000e-3.db}
target_library = {xprim_4010e-3.db xprim_4000e-3.db xgen_4000e.db xfpga_4000e-3.db
xio_4000e-3.db}
define_design_lib xdw_4000e -path /CMC/tools/xilinx.vM1.3/synopsys/libraries/dw/lib/xc4000e
symbol_library = {xc4000e.sdb}
synthetic_library = {xdw_4000e.sldb standard.sldb}

ted@dea ~/SYNOPSYS 11:48am >
```

II. Performing Logic Synthesis

This section will explain the use of the Synopsys tools used to perform logic synthesis. We will primarily explore the use of Design Compiler shell scripts. These are ASCII text files which are invoked from the command line with the **dc_shell** command. The other alternative is the use the Synopsys Design Analyzer to perform synthesis using the graphical user interface. Shell scripts are convenient in that they do not require access to a graphics terminal, they may be executed over a modem connection at one's convenience. You will find that a few simple changes to an existing script will allow you to synthesize a different design. Most of the commands are the same, all that is necessary is to change some of your input file names and output file names.

If you have not yet done so, source the synopsys.env file to set up your environment to allow you to run the dc_shell command. You will also have to create a new directories in your Synopsys directory as explained below.

(1) From your Synopsys directory create a subdirectory Synthesized and a subdirectory called XNF. The Synthesized directory will hold the results of the synthesis procedure in a Synopsys database (db) format. The XNF directory will contain the Xilinx netlist files generated by the synthesis procedure. Create a subdirectory called Scripts; this will be used to hold the various shell scripts we will be writing. These steps may be performed by issueing the following UNIX commands:

```
% cd          (this will return you to your home directory)
% cd Synopsys              (change to your Synopsys directory)
% mkdir Synthesized
% mkdir XNF
% mkdir Scripts
```

(2) We will be executing the dc_shell command from the Synopsys directory. Our shell scripts will be analyzing VHDL files, to avoid having our Synopsys directory cluttered up intermediate work files we will tell Synopsys to store any intermediate files generated in the ./Code/Work directory. Create a file called .synopsys_vss.setup in the Synopsys directory with the following contents:

```
WORK > DEFAULT
DEFAULT: ./Code/Work
TIMEBASE = NS
```

Although it is not essentially necessary to have this file, it will avoid having intermediate files stored in the main Synopsys directory.

Example 1: Synthesizing a half-adder circuit.

(1) Change into your Code directory and create a file called half_adder.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 concurrent of half_adder is
begin
  carry <= not (in1 and in2);  -- active low outputs
  sum   <= not (in1 xor in2);  -- for board
end concurrent;
```

(2) Change into your Scripts directory and create a file called half_adder.scr with the following contents:

```
/* Script to analyze and elaborate */
/*  half adder using  concurrent assignments      */
/* Ted Obuchowicz                  */

analyze -format vhdl ./Code/half_adder.vhd
elaborate half_adder
set_max_area 0
current_design half_adder
set_port_is_pad “*”
insert_pads -verify -verify_effort low
compile -map_effort high -verify
write -format db -hierarchy -output ./Synthesized/half_adder_before_replace_fpga.db

/*replace the CLBs and IOBs with gates */

replace_fpga;

/* set part number */

set_attribute half_adder “part” -type string  “4010epc84-3”

/* add pin locations  */

set_attribute “in1” “pad_location” -type string “P19”
set_attribute “in2” “pad_location” -type string “P20”
set_attribute “carry” “pad_location” -type string “P61”
set_attribute “sum” “pad_location” -type string “P62”


/* write to a .db post replace fpga */

write -format db -hierarchy -output ./Synthesized/half_adder.db
write -format xnf -hierarchy -output ./XNF/half_adder.xnf
quit
```

Let’s take a few minutes to explain this script file. The first three lines are examples of comments. Comments are enclosed in the /* and */ delimiter pairs. The next line analyzes the input file found in the ./Code/half_adder.vhd directory. Note: this script was meant to be executed from the Synopsys directory, the . in the filename is UNIX shorthand for the present working directory.

This example consists of a single VHDL source file; scripts written for designs which consist of multiple source files should analyze the files one by one from the bottom up.

The elaborate half_adder line builds the specified design from the intermediate files stored in the work library. The design name is specified as the top-level VHDL entity, in this case the entity name is half_adder. Recall, all the intermediate files are stored in the ./Code/Work directory. The tool must know where to look for these intermediate files, hence the need for the .synopsys_vss.setup file in the directory from which the dc_shell command was invoked from. If there were no setup file in this directory, the tool would store any intermediate files in this directory and would know to look there for them when it came time to elaborate the design.

The set_max_area 0 line tells the compiler to create a design which will be optimized for the smallest possible size.

The next line current_design half_adder tells the Design Compiler that the following commands should apply to this entity name.

We want to have all the ports in the top-level entity (half_adder) to be associated with input/output pads of our target architecture. The set_port_is_pad "*" command will tell the Design Compiler that all the ports in the top-level entity half_adder are to have I/O pads attached to them. The actual pads will be inserted by the insert_pads command which follows.

The insert_pads -verify -verify_effort low command will add a XILINX Input/Output Block (IOB) to each port which has had the port_is_pad attribute set on it by a preceding set_port_is_pad command. The -verify -verify_effort low tells the Design Compiler to perform a functional comparison between the initial design and the padded result without spending too many CPU cycles doing so.

The compile command performs logic and gate level synthesis on the current design.

The next command tells the Design Compiler to store the result of the synthesis in a special Synopsys database format in the ./Synthesized directory with filename half_adder_before_replace_fpga.db. At this point, our synthesized design will consist of Xilinx IOBs and CLB's (Configurable Logic Blocks).

The replace_fpga command is used to replace field-programmable cells in the current design with logic gates contained in the target library.

The rest of the script then tells the compiler which device we are targetting, associates the ports of the entity with actual device pin numbers, writes out the gate-level synthesized design to a half_adder.db file in the Synthesized directory, generates a Xilinx netlist file with name half_adder.xnf, and finally quits. Don't concern yourself too much with how the pin numbers were chosen; this information is given in Part III of this tutorial.

The next step is to run the script using the dc_shell command.

(3) Change into your Synopsys directory and invoke the command

% dc_shell -f ./Scripts/half_adder.scr

This will invoke the Design Compiler shell and execute the commands contained in the half_adder.scr file. The Design Compiler will run and periodically report its progress. Make sure that it terminates without any errors. If there are errors, check your half_adder.scr file for possible typing errors etc.

(4) We will use the Design analyzer to examine the pre and post replace_fpga synthesis results. type design_analyzer from the Unix command (enter this command from the Synopsys directory). Select File -> Read from the top portion of the Synopsys Design Analyzer window. The Read File window will appear as shown in Figure 14. From this window, left click on the Synthesized directory in the central portion of the window. After doing so, the Synthesized/ directory will appear in the File Name(s) field. Move the cursor to the right of this name and left click the mouse. Press the Return key on the keyboard. The scroll window will now list the files found your Synthesized directory (see Figure 15). Select half_adder_before_replace_fpga.db and select the OK button.

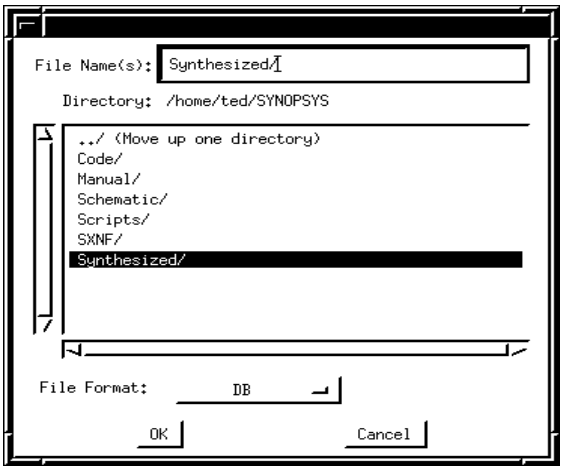

Figure 14: Read File.

In the central portion of the Design analyzer window will be a yellow square labelled half_adder (see Figure 16). Select this square with the left mouse button, it will become outlined in a dashed line. Descend down the hierarchy by selecting the down arrow button, the symbol view of the half_adder entity will be shown listing input and output ports (see Figure 17). Select the button with the AND gate symbol, the schematic in terms of Xilinx IOB's and CLBs will be shown. In this design there are 4 IOB's and one CLB (see Figure 18).

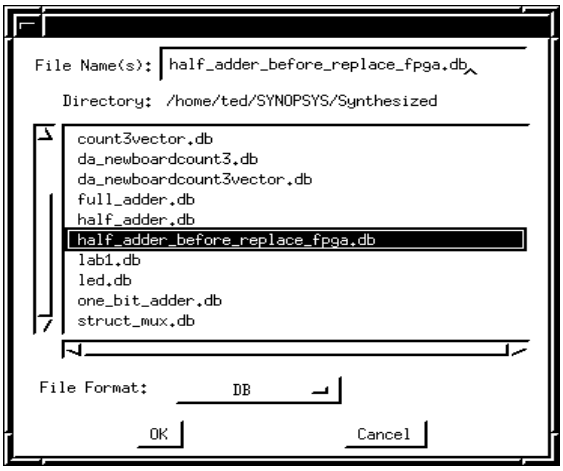

Figure 15: Selecting a design.

Figure 16: Half_adder entity.

Figure 17: Symbol view of half_adder entity showing ports.

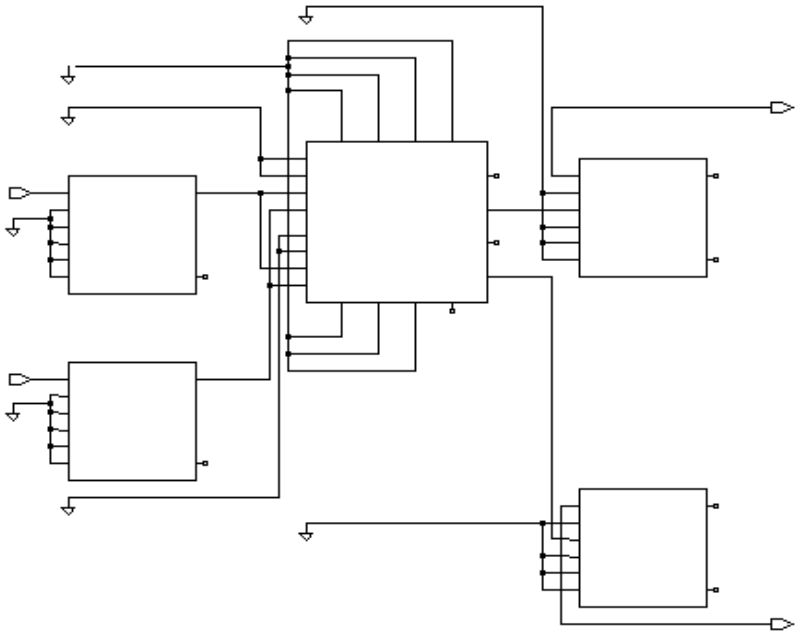

Figure 18: Schematic of half_adder in terms of Xilinx blocks.

(5) Use the Design Analyzer to read in the half_adder.db file from the Synthesized directory. Descend down the design hierarchy and view the schematic. What are the differences between the two synthesized designs? You can see the effect that the replace_fpga command had on the two results. The post replace_fpga design now consists of a gate level schematic diagram. The buffers labelled IBUF and OBUF_S are input/output buffers added by the synthesis tool. See Figure 19.

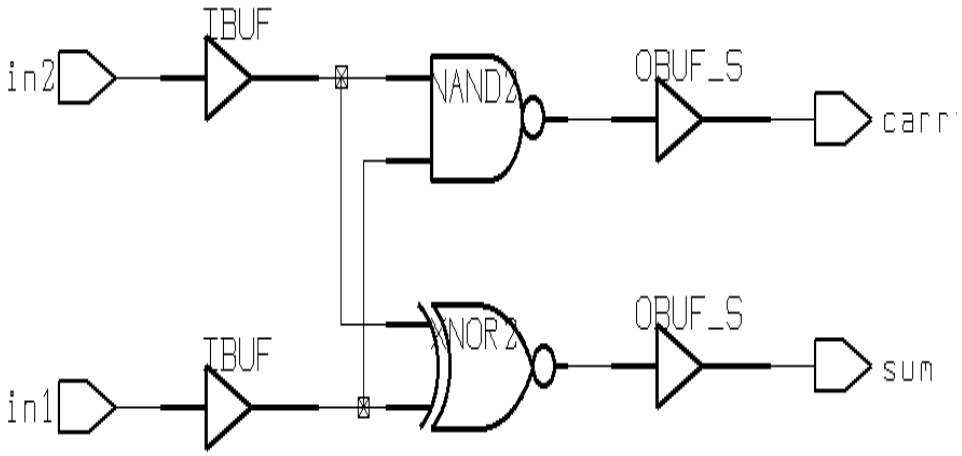

Figure 19: Post replace_fpga synthesis results for half_adder design.

Example 2: Synthesizing a structural VHDL design.

This example will introduce a new dc_shell command: **uniquify**. This command must be used when working with structural VHDL which consist of multiple instances of similar components. The example consists of a full adder circuit constructed from two half adders and an OR gate. There is some additional code which serves to decode the SUM and CARRY_OUT signals to drive a 7-segment LED display. The result of the binary full addition is displayed in decimal on the LED display.

(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.vhd with the following contents:

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

entity full_adder is
  port(carry_in, input1, input2 : in std_logic;
       output : out std_logic_vector(6 downto 0));
end full_adder;

architecture structural of full_adder 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 sum_out, carry_out : std_logic ;
signal carry1, carry2     : std_logic;
signal sum_int            : std_logic;

-- declare configuration specification
-- NOTE: we want to use the half adder with true outputs
-- not the inverted ones we synthesized earlier!!

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;


output <= “0000001” when ( (sum_out = ‘0’) and (carry_out = ‘0’) ) else
          “1001111” when ( (sum_out = ‘1’) and (carry_out = ‘0’) ) else
          “0010010” when ( (sum_out = ‘0’) and (carry_out = ‘1’) ) else
          “0000110” when ( (sum_out = ‘1’) and (carry_out =’1’) )  else
          “1111111” ;

end structural;
```

Note the line containing the configuration specification. For each instance of the half_adder component we want the tool to use the architecture which has the “true” (non-inverted) outputs. Note how easy it is to select a particular architecture associated with an entity. VHDL allows an entity to have more than one architecture associated with it. The configuration specification is a mechanism whereby a particular architecture is chosen for a particular instantiation of a component.

(2) In your Scripts directory create a full_adder.scr file:

```
/* Script to analyze and elaborate */
/*  fulladder       */
/* Ted Obuchowicz                  */
analyze -format vhdl ./Code/half_adder_regular_outputs.vhd
analyze -format vhdl ./Code/full_adder.vhd
elaborate full_adder
set_max_area 0
current_design full_adder
uniquify
set_port_is_pad “*”
insert_pads -verify -verify_effort low
compile -map_effort high -verify
write -format db -hierarchy -output ./Synthesized/full_adder_before_replace_fpga.db

/*replace the CLBs and IOBs with gates */

replace_fpga;

/* set part number */

set_attribute full_adder “part” -type string  “4010epc84-3”

/* add pin locations  */

set_attribute “carry_in” “pad_location” -type string “P19”
set_attribute “input1” “pad_location” -type string “P20”
set_attribute “input2” “pad_location” -type string “P23”
set_attribute “output<6>” “pad_location” -type string “P49”
set_attribute “output<5>” “pad_location” -type string “P48”
set_attribute “output<4>” “pad_location” -type string “P47”
set_attribute “output<3>” “pad_location” -type string “P46”
set_attribute “output<2>” “pad_location” -type string “P45”
set_attribute “output<1>” “pad_location” -type string “P50”
set_attribute “output<0>” “pad_location” -type string “P51”
/* write to a .db post replace fpga */
```

```
write -format db -hierarchy -output ./Synthesized/full_adder.db
/* write out the Xilinx netlist file */
write -format xnf -hierarchy -output ./XNF/full_adder.xnf
quit
```

This script is similar to the half_adder script. The main difference is to use of the uniquify command. This command is used to Removes multiply-instantiated hierarchy in the current_design by creating a unique design for each cell instance. It does this by appending an integer to the component name for every instantiation statement in the source code. The uniquify command may represent a substantial amount of total execution time during synthesis. For this reason, try to keep structural VHDL only in your top-level entity.

(3) Execute the full_adder script from the Synopsys directory by entering:

% dc_shell -f ./Scripts/full_adder.scr

(4) Invoke the Design analyzer and read in the full_adder_before_replace_fpga.db synthesized design. Notice how each instance of the half_adder component has been “uniquified”; each instance has been given a unique name: half_adder_0 and half_adder_1. See Figure 20.

Figure 20: Uniquified full_adder design.

(5) Descend down into the hierarchy of the half_adder_0 component by selecting it and using the down arrow button. Notice how this component is implemented in a single Xilinx CLB. The half_adder_1 component is similarly implemented.

(6) Return to the top-level (use the up arrow button) and descend down into the full_adder hierarchy. Notice how it consists of IOBs, CLBs, and two half_adder components. See Figure 21.

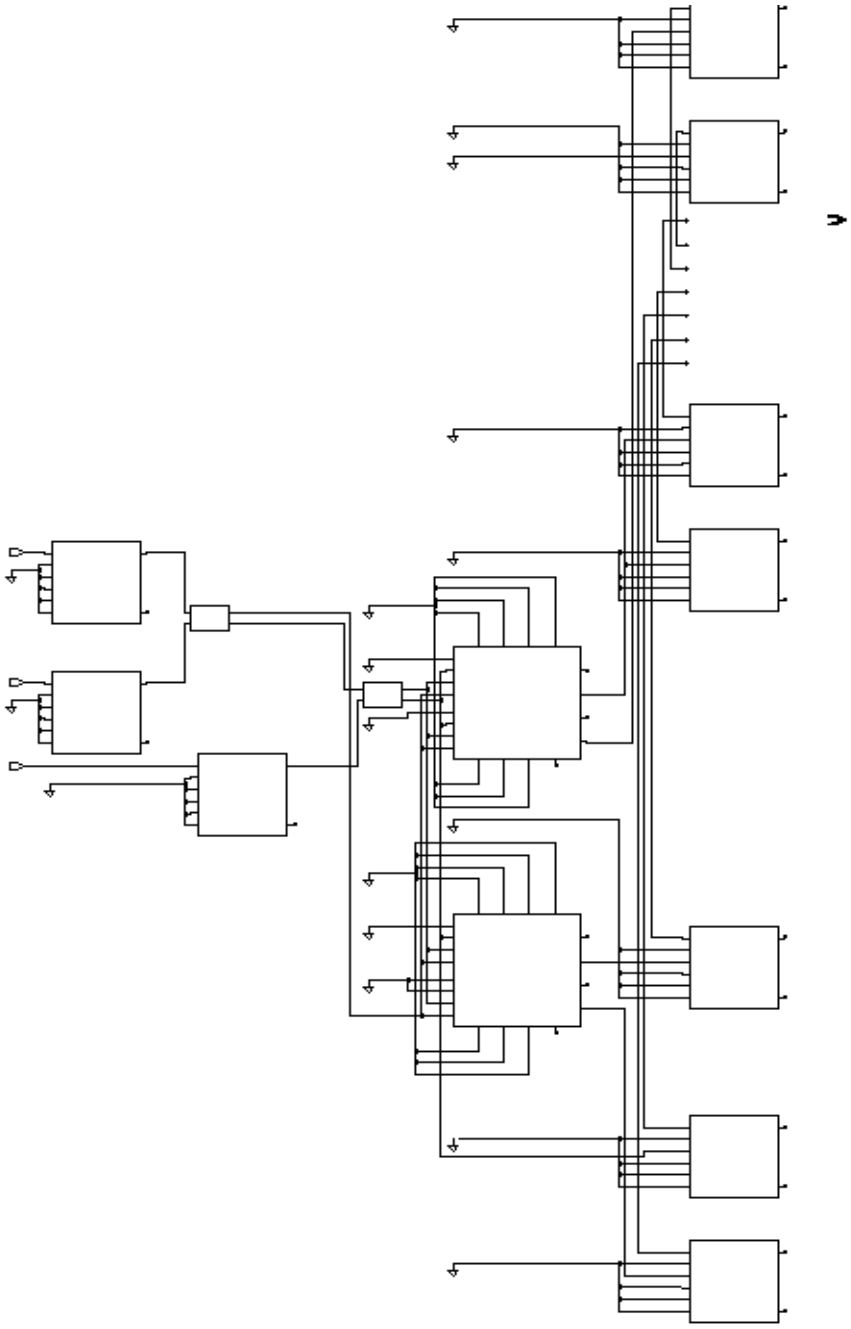

Figure 21: Full_adder schematic in terms of Xilinx blocks.

(7) Use the Design Analyzer to examine the full_adder.db synthesized design. This is the result of synthesis after FPGA cells have been replaced with gates form the technology library. Select File -> Read -> Synthesized/full_adder.db and descend down the hierarchy of this design. Traverse down the the gate level schematic (see Figure 22). Notice how each of the IOB in Figure 21 have been replaced with either an IBUF or an OBUF_S. Notice also that the two half adders

are represented by yellow squares. You can descend down into the hierarchy of either half_adder by selecting it (it's outline will change from a solid line to a dashed line) and selecting the down arrow button of the Design Analyzer. Notice how the gate-level schematic of each half adder does not have inverted outputs. Finally, we can see that the two CLBs in the middle of Figure 21 have been replaced with a combinational logic network.

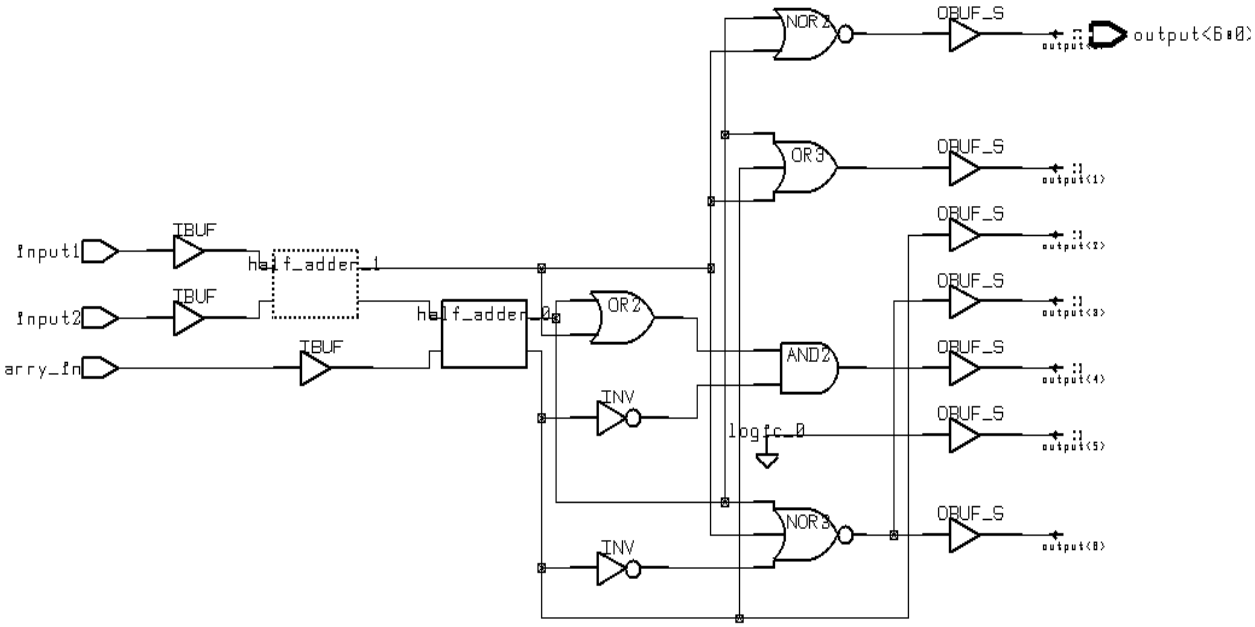

Figure 22: Full adder schematic after replacing FPGA cells with gates.

Example 3: 3-bit binary counter.

This example synthesizes a 3-bit binary counter. The script file illustrates the syntax adopted by Synopsys for naming elements of vectors. It also introduces a new command: **report_fpga**. This command is issued prior to the replace_fpga command. The report_fpga command is used to generate a report about FPGA resource usage. Xilinx cell information includes CLB and IOB statistics, as well as information about other cell resources. For CLBs, the number of F, G and H function generators are listed as well as the number of CLB's. IO information includes number of ports, number of Clock Pads (such as BUFGS cells), and the number of IOB cells used.

(1) In your Code subdirectory create a file called count3_vector.vhd with the following contents:

```
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;
```

(2) In your Scripts directory, edit a file named count3_vector.scr with the following commands in it:

```
/* Script to analyze and elaborate */
/* 3 bit counter                   */
/* Ted Obuchowicz                  */
/* VHDL file uses std_logic_vector */

analyze -format vhdl ./Code/count3_vector.vhd
elaborate count3
set_max_area 0
current_design count3
set_port_is_pad “*”
insert_pads -verify -verify_effort low
compile -map_effort high -verify
write -format db -hierarchy -output ./Synthesized/count3vector_before_replace_fpga.db
/* generate an FPGA resource usage report before replace_fpga */
report_fpga

/*replace the CLBs and IOBs with gates */
```

```
replace_fpga

/* set part number */

set_attribute count3 “part” -type string  “4010epc84-3”

/* set the input-output pin locations  */

set_attribute “clk” “pad_location” -type string “P10”
set_attribute “resetn” “pad_location” -type string “P19”
set_attribute “count_en” “pad_location” -type string “P28”
set_attribute “cout” “pad_location” -type string “P61”
set_attribute “sum<2>” “pad_location” -type  string “P58”
set_attribute “sum<1>”  “pad_location” -type string  “P59”
set_attribute “sum<0>”  “pad_location” -type string  “P60”

/* write to a .db post replace fpga */

write -format db -hierarchy -output ./Synthesized/count3vector.db

write -format xnf -hierarchy -output ./XNF/count3vector.xnf
quit
```

Note the use of the < and >, as in sum<1>, to refer to individual elements of a vector data type.

(3) Use the dc_shell to run the count3_vector.scr:

% dc_shell -f ./Scripts/count3_vector.scr

After the elaborate count3 command is executed, the Design Compiler will report that it “inferred memory devices”. This means the tool recognized the need for flip-flops in the synthesized design. Since this design was a 3-bit binary counter, 3 memory elements were inferred:

```
Inferred memory devices in process
        in routine count3 line 15 in file
         ‘/home/ted/SYNOPSYS/Code/count3_vector.vhd’.
===============================================================================
|      Register Name       |   Type    | Width | Bus | AR | AS | SR | SS | ST |
===============================================================================
|        count_reg         | Flip-flop |   3   |  Y  | Y  | N  | N  | N  | N  |
===============================================================================
```

When the script reaches the report_fpga command, the following will be listed in the window from which the script is executing in:

```
****************************************
Report : fpga
Design : count3
Version: v3.4b
Date   : Thu Jul 23 13:26:58 1998
****************************************

  Xilinx FPGA Design Statistics
```

```
-----------------------------

  FG Function Generators:            3
  H Function Generators:             0
  Number of CLB cells:               3
  Number of Hard Macros and
      Other Cells:                   1
  Number of CLBs in
      Other Cells:                   3
  Total Number of CLBs:              6

  Number of Ports:                   7
  Number of Clock Pads:              1
  Number of IOBs:                    6

  Number of Flip Flops:              3
  Number of 3-State Buffers:         0

  Total Number of Cells:            11
```

(4) Use the Design analyzer to view the two results (before and after replace_fpga). Can you identify the fpga resources used? Figures 23 and 24 give the results of the synthesis.

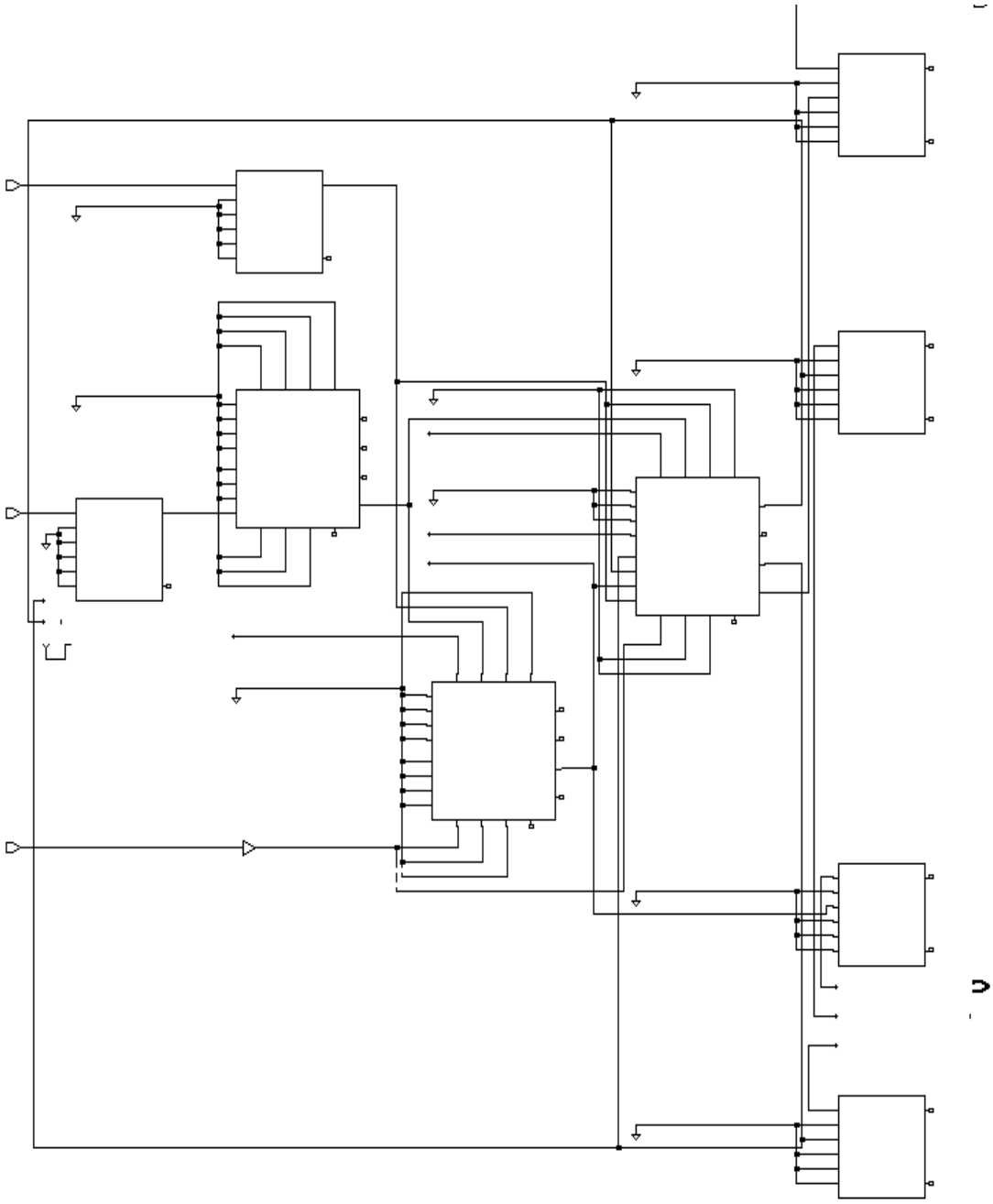

Figure 23: Count3 design before replace_fpga.

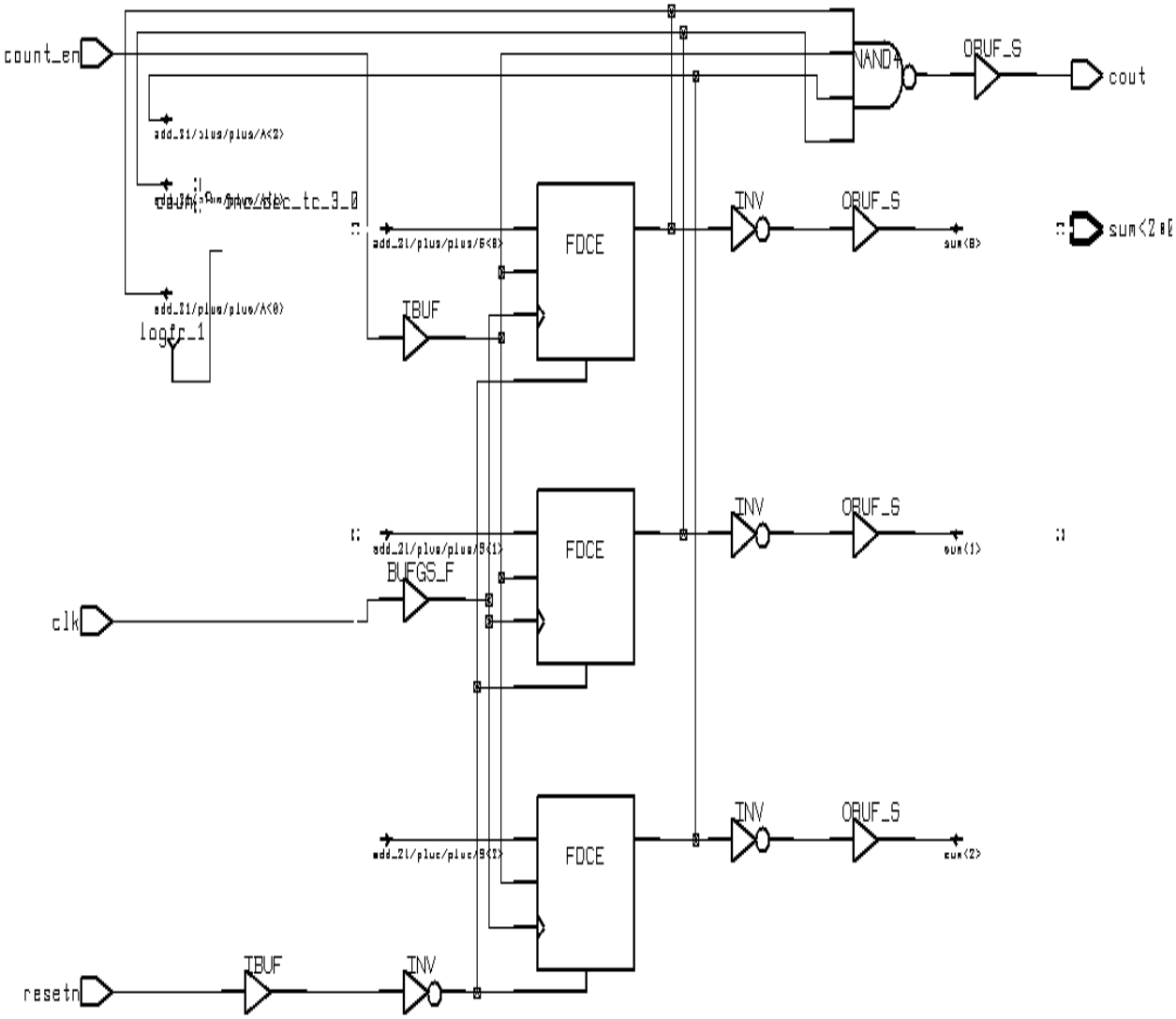

Figure 24: Count3 design after replace_fpga.

PART III : Implementation using Xilinx Design Manager

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

(i) the XNF 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 program the FPGA on the demonstration board via the serial port of the workstation.

I. Setting up the user environment to run the Xilinx Design Manager program

Prior to using the Xilinx tools, source the file /home/ted/ENVIRONMENT/xilinx.env. Also note that the current version (3.4b) of this tool only works with the Solaris 2.5.1 operating system (for version 1998.08-1 of Synopsys this is no longer a problem) . To find out the version of operating system running on your workstation, issue the UNIX command “uname -r”:

```
ted@fbi ~ 12:34pm >uname -r
5.5.1
ted@fbi ~ 12:34pm >
```

uname will report 5.5.1 for Solaris 2.5.1 machines. If you are not logged onto a Solaris 2.5.1 machine perform the following:

```
% echo $DISPLAY        (remember the name of the DISPLAY which will be reported)
% rlogin fbi
$ setenv DISPLAY name:0.0  (where name was the name reported by the echo command)
```

It will be necessary to source the xilinx.env file from the machine you performed the remote login to.

II. Implementing a Design with the Xilinx Design Manager

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

(2) Invoke the Xilinx Design Manager with the following command:

% dsgnmgr &

after some time the Design Manager window will appear as shown in Figure 25.

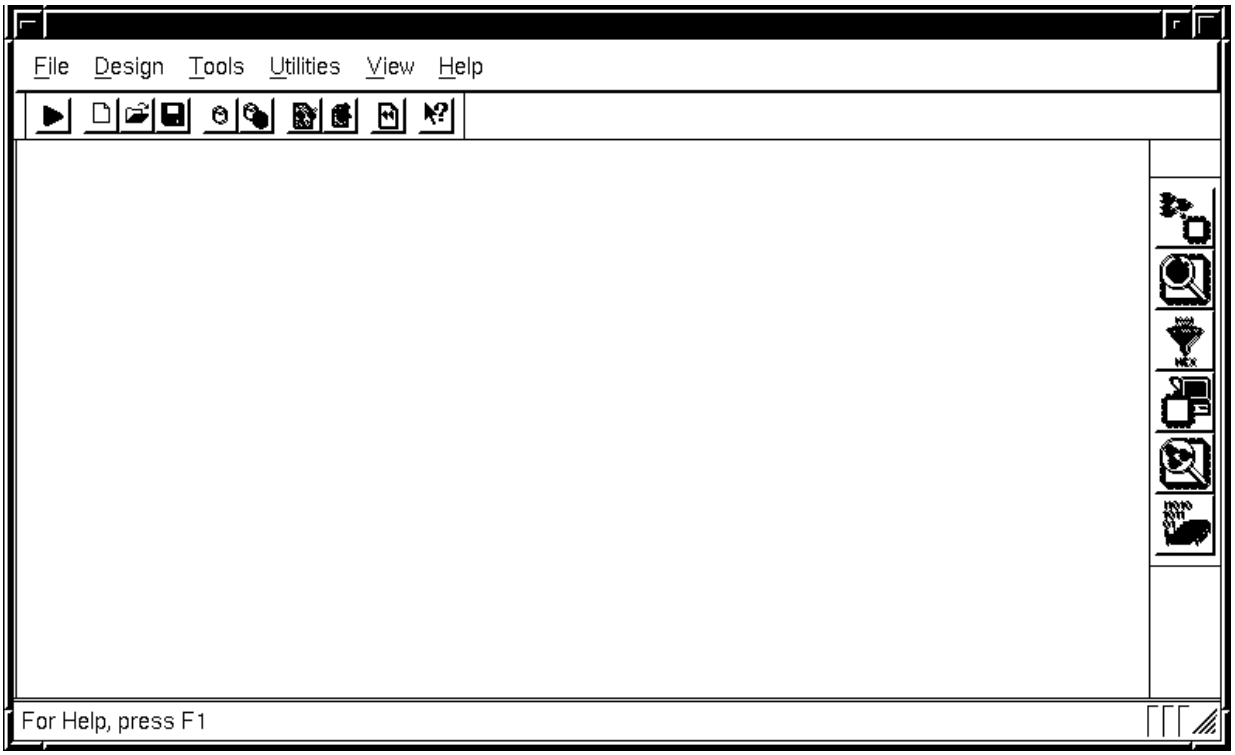

Figure 25: Xilinx Design Manager.

(3) Create a new Project by selecting File -> New Project from the Design Manager top menu bar. The New Project window will appear (see Figure 26). In this window fill in the following fields:

Input Design: /home/user_name/Synopsys/XNF/half_adder.xnf
Working Directory: /home/user_name/Synopsys/Xilinx/halfadder

Leave the Comment field empty, it is optional. Select the OK button. Be sure the substitute your actual login name in place of “user_name” listed in the example given above.

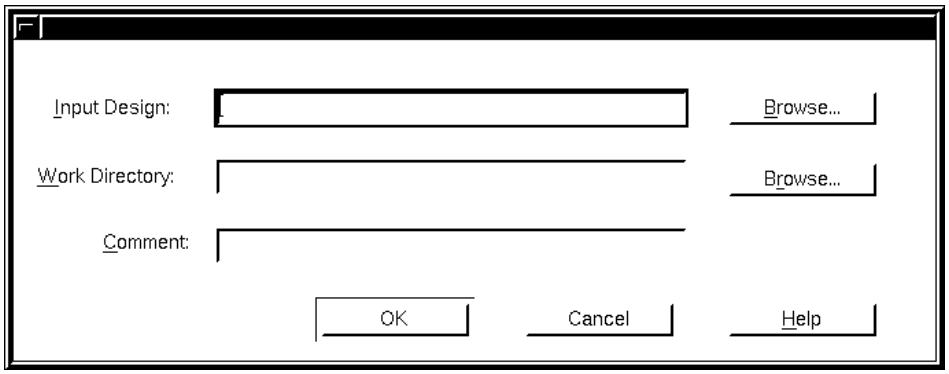

Figure 26: New Project window.

(4) The Design Manager window will now list a new project in the central portion of the window. Select Design -> Implement from this window. Select the required part (XC4010-3-PC84) by clicking on the Select Button and filling in the Part Selector form as shown in Figure 28. To obtain a list of possible choices, select the small button marked with a traingle to the left of the field.

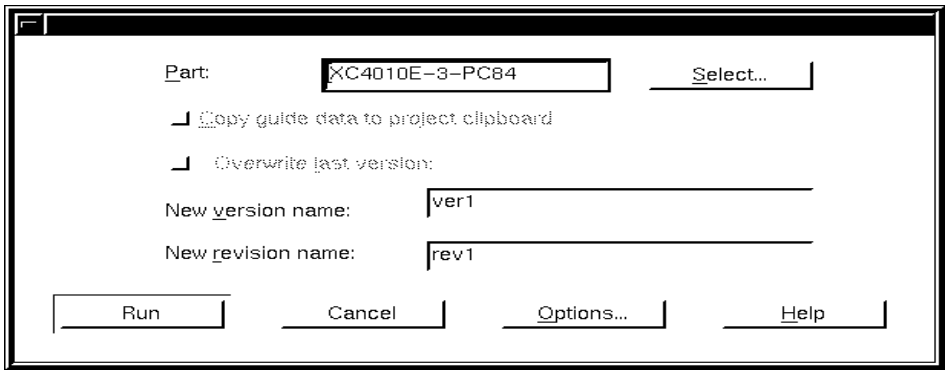

Figure 27 : Implement Form.

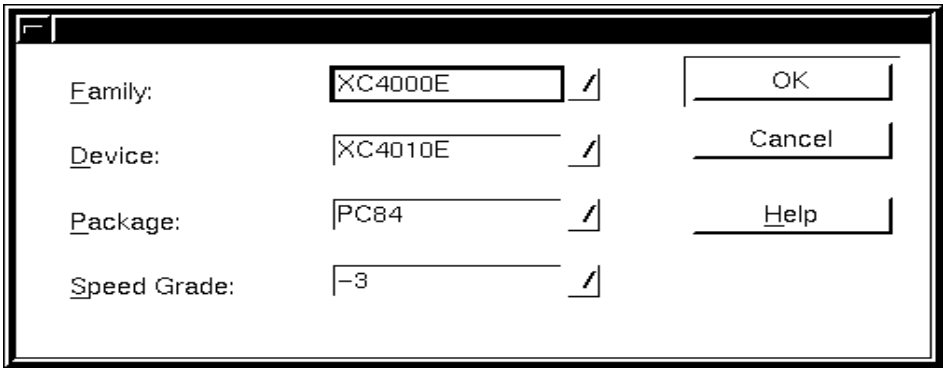

Figure 28: Part Selector.

(5) From the Implement form, select the Options button at the bottom of the form. Fill out the form as given in Figure 29. We are interested in having only Configuration Data produced, leave the other fields to their default values. Select OK from the Options form.

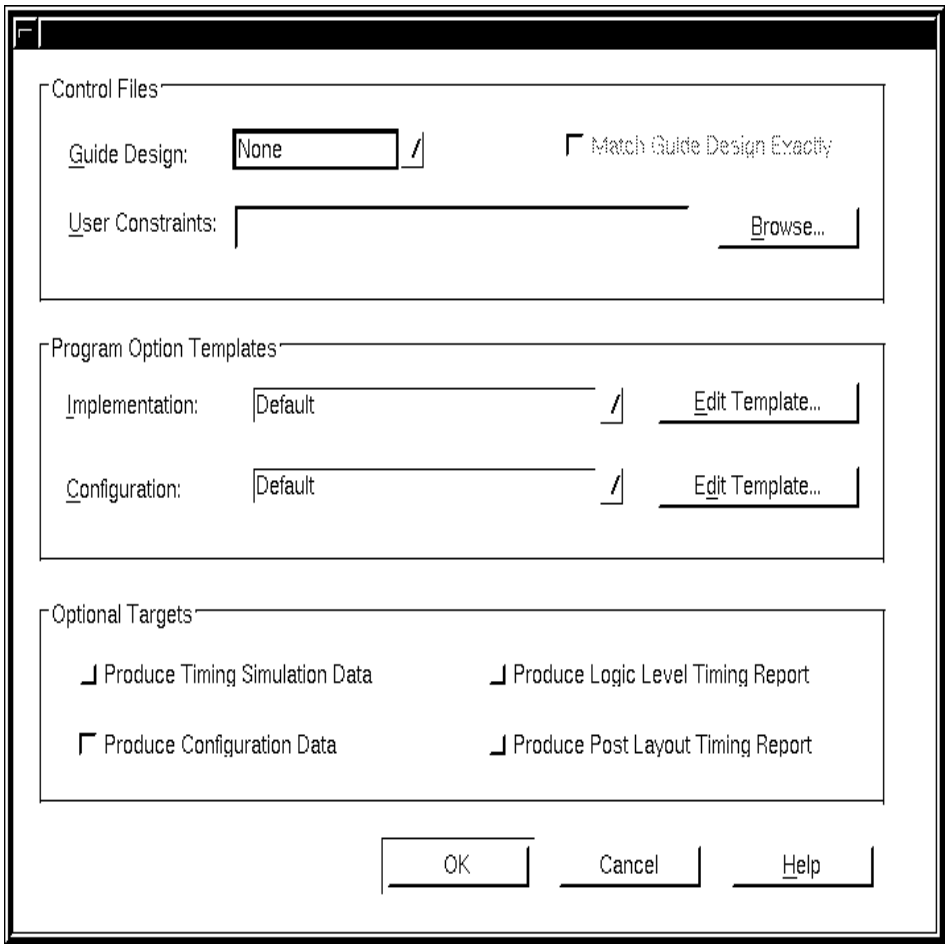

Figure 29: Setting the Options form.

(6) Once you have set the options, select Run from the Implement form. The Flow Engine window will appear (see Figure 30) and show the progress of the four steps involved in the implementation: **Translate**, **Map**, **Place and Route**, **Configure**. Certain stages may take long to complete. At the end of implementation, the Flow Engine will report that the design was succesfully implemented. There should be a file called half_adder.bit in the Xilinx directory at this point. The next step is to download the configuration file to the demo board and verify the working hardware.

(7) Save the half_adder project by selecting File -> Save Project from the Design Manager window.

NOTE: If you are using version M1.5 of the Xilinx Alliance software it will be necessary to

change to default place and route effort level from 2 to 4 (or higher). From the above Options form select Edit Template (next to the Implementation field in the Program Option Templates), then select Place and Route at the top of the popup window. Change the slider control from 2 to 4. Click OK in each window to return to the Implementation window.

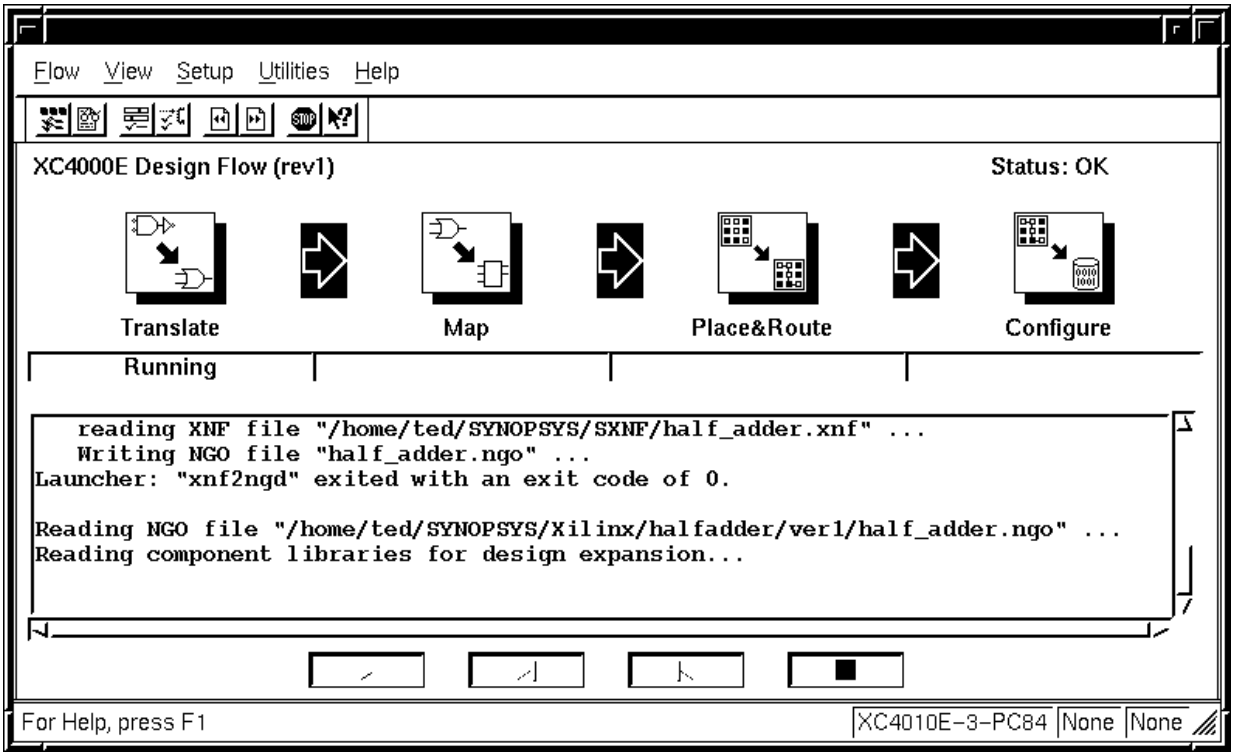

Figure 30: Flow Engine in progress.

III. Downloading a design to the demonstration board

The Hardware Debugger will be used to download the bit file to the FPGA demonstration board. Once downloaded, the hardware may be tested. Configuration data is retained in the FPGA only as long as power is applied to the board. Since the FPGA is a lookup table architecture, any data will be lost when power is turned off.

(1) At this point, you must be using a workstation which has the required Xchecker cable connecting the workstation's serial port and the demo board. With the half_adder project still open, select Tools -> Hardware Debugger from the Design Manager's menu bar. The Hardware Debugger window will appear as shown in Figure 31. An additional window will appear (see Figure 32) with the notice that the design does not have a READBACK block connected. This is normal. Select OK in this window to continue.

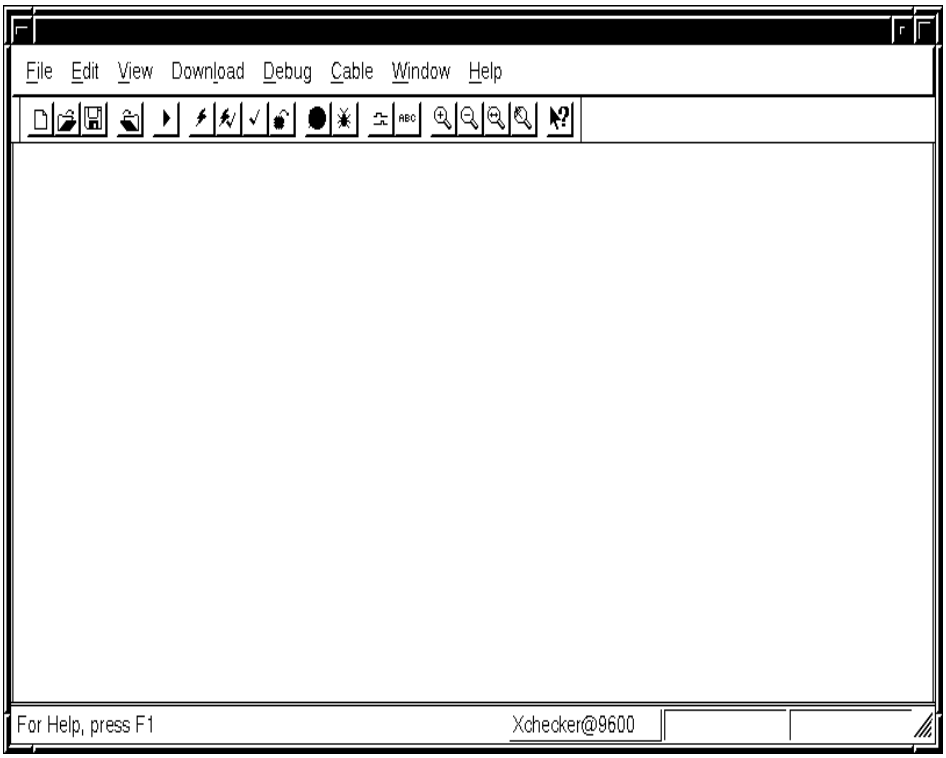

Figure 31: Hardware Debugger.

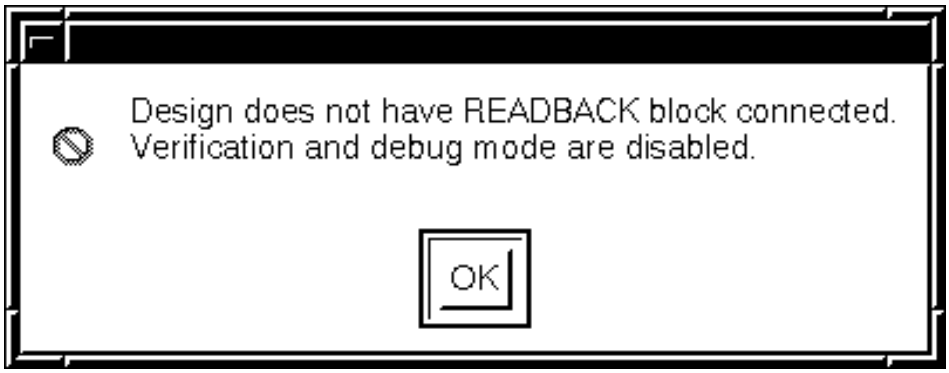

Figure 32: Verification and Debug notice.

(2) The Hardware Debugger window will now list the current design (half_adder). Use the mouse to highlight the half_adder.bit file listed under DESIGNS (see Figure 33). After selecting the bit file, select Download from the top menu bar.

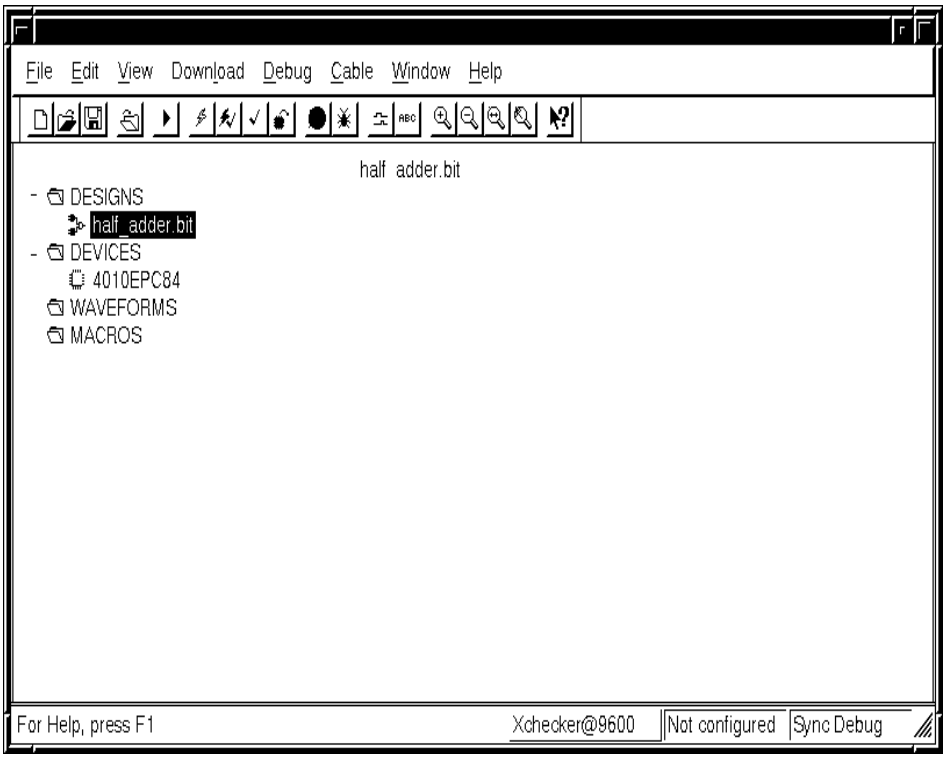

Figure 33: Selecting the half_adder.bit file for download.

(3) A small window (see Figure 34) showing the progress of the device download will appear. When the bit file has been transferred to the board, a window will appear indicating the device download is complete (Figure 35). Select OK in this window.

(4) You may now test the design using the demonstration board. Set the DIP switches (located in the centre of the board) to different combinations and verify the Carry and Sum outputs as indicated on the LED bar display.

(5) Quit the Hardware Debugger (File -> Exit) and quit the Design Manager.

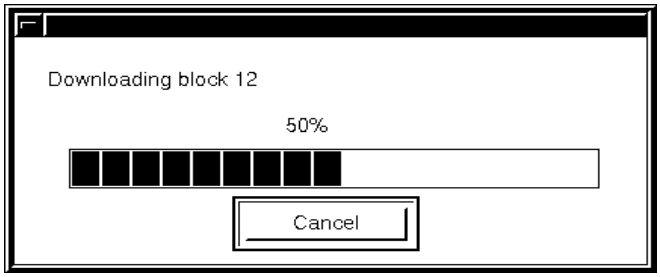

Figure 34: Device download progress window.

Figure 35: Download complete window.

PART IV : Xilinx FPGA Demonstration Board

The Xilinx FPGA demonstration board (Figure 36) is used to program and testing Xilinx FPGAs (XC3000 and XC4000 devices) using the Xilinx Alliance Series software (Design Manager).

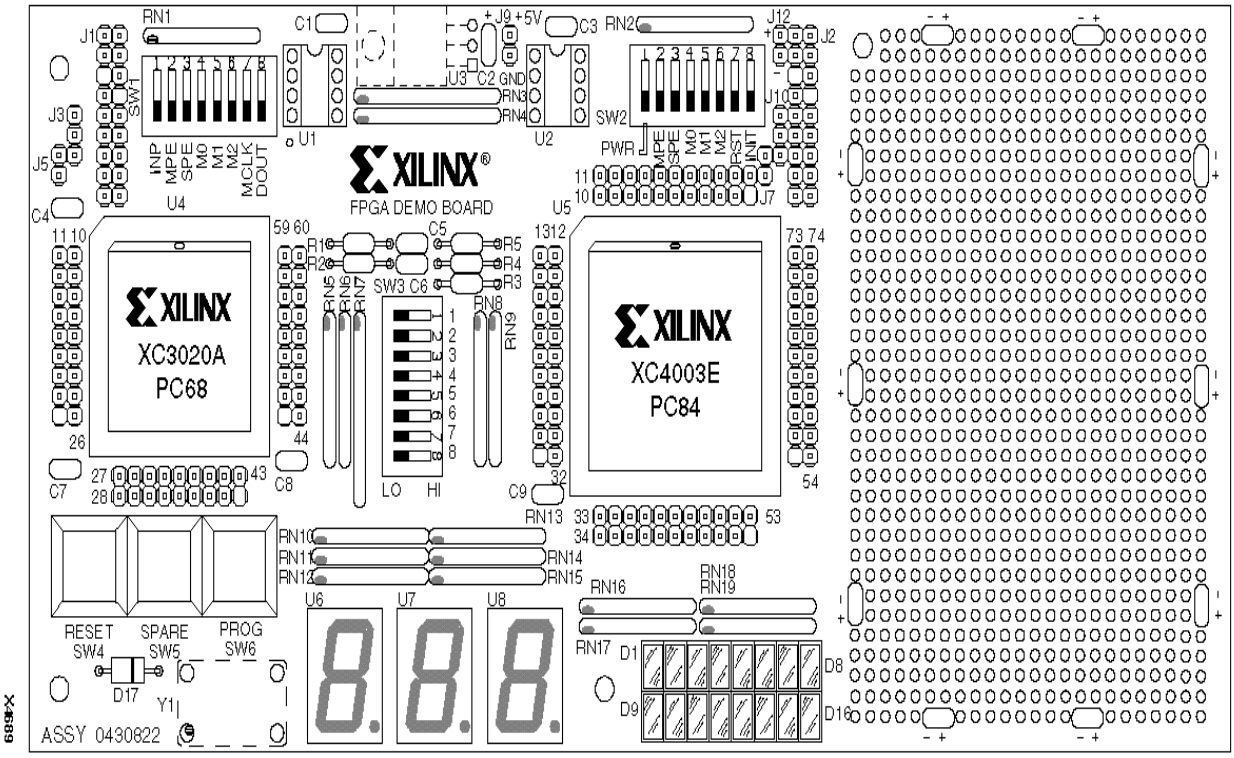

Figure 36: Xilinx demonstration board layout.

The board is populated with two Xilinx FPGAs: a XC3020A PC68 and a XC4010E PC84. In addition to these programmable devices, the board contains a series of DIP switches, LED bar indicators, and LED 7-segment displays. Note that the FPGA chips are volatile; they will need to be reprogrammed every time the power supply to the bouard is turned off. The remainder of this section describes the demonstration's boards components used in programming and testing.

I. General Purpose Input Switches (SW3)

This DIP switch provides 8 general purpose inputs to both FPGAs on the demo board. The switch is located vertically between the two FPGAs. When a switch is in the ON position (pressed inwards towards the switch number on the right-hand side of the board), a logic "1" is applied to the FPGA's pin. A logic "0" is applied when the switch is in the OFF position. Table 1 give the pin connections for SW3.

Table 1: SW3 Pin Connections

Switch SW3	Pin number on XC3020A	Pin number on XC4010E
1	11	19
2	13	20
3	15	23
4	17	24
5	19	25
6	21	26
7	23	27
8	24	28

II. 7-Segment Display (U6, U7, U8)

The XC4010E chip drives the two right most 7-segment displays. The left display is connected to the XC3020A chip. These 7-segment display units are **ACTIVE LOW**. This means that the corresponding segment will be lit when a **logic "0"** is applied to its pin number. The pin connections for the three 7-segment displays are given in Table 2 and the segments of the display are shown in Figure 37.

Table 2: Pin connections for 7-segment displays

7-Segment	XC3020A	XC4010E	XC4010E
Display	U6	U7	U8
a	38	39	49
b	39	38	48
c	40	36	47
d	56	35	46
e	49	29	45
f	53	40	50

Table 2: Pin connections for 7-segment displays

7-Segment	XC3020A	XC4010E	XC4010E
g	55	44	51
decimal point	30	37	41

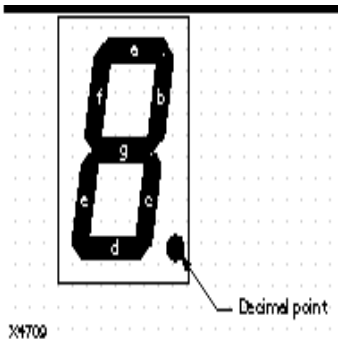

Figure 37: 7-Segment display LED segments.

The decimal point of the rightmost display (U8) is connected to pin 41 of the XC4010E and acts as a programming error indicator. During programming of the chip, this decimal point should not be lit. If it comes back on, this indicates a programming error. The decimal points of U6 and U7 are tied low during programming and are on when the FPGAs are waiting to be programmed.

III. LED Bar Indicators (D1-D8, D9-D16)

There are two LED Bar indicators on the demonstration board. They are located on the right-hand side of the board below the XC4010E chip. The top bar indicator is connected to the XC3020A, the lower 8 LEDs are connected to the pins of the XC4010E. These LEDs are also **ACTIVE LOW**. Table 3 gives the pin connections for the LED bar indicators.

Table 3: LED Bar Indicator Connections

Top LEDs	XC3020 pin numbers	Bottom LEDs	XC4010E pin numbers
D1	37	D9	61
D2	36	D10	62

Table 3: LED Bar Indicator Connections

Top LEDs	XC3020 pin numbers	Bottom LEDs	XC4010E pin numbers
D3	41	D11	65
D4	33	D12	66
D5	32	D13	57
D6	31	D14	58
D7	28	D15	59
D8	29	D16	60

IV. Expansion Input/Output

The Xilinx FPGA demonstration boards available in the lab have been modified by the ECE technical staff to include 8 additional DIP switch inputs and 8 additional BAR LED outputs. Tables 4 and 5 give the pin locations for the inputs and outputs respectively.

Table 4: Expansion DIP Switch Inputs

SWITCH	XC4010E pin number
SW1	9
SW2	8
SW3	7
SW4	6
SW5	5
SW6	4
SW7	3
SW8	84

Table 5: Expansion BAR LED Indicators

LED	XC4010E pin number
D1	83
D2	82
D3	81
D4	80
D5	79
D6	78
D7	77
D8	70

V. More information on the FPGA Demonstration Board

More information regarding the Xilinx FPGA demonstration board can be found using the online Xilinx documentation, available through the Dynatext browser. To use the Dynatext browser type **dtext &** from the UNIX prompt (make sure you have sourced the xilinx.env file prior to invoking the browser). The Dynatext window will appear (Figure 38). Select Xilinx Books. A new window will appear. Double click on the Hardware User Guide title to open this document. A hardcopy of Chapter 1 of the Hardware User Guide is included in the Appendix for convenience.

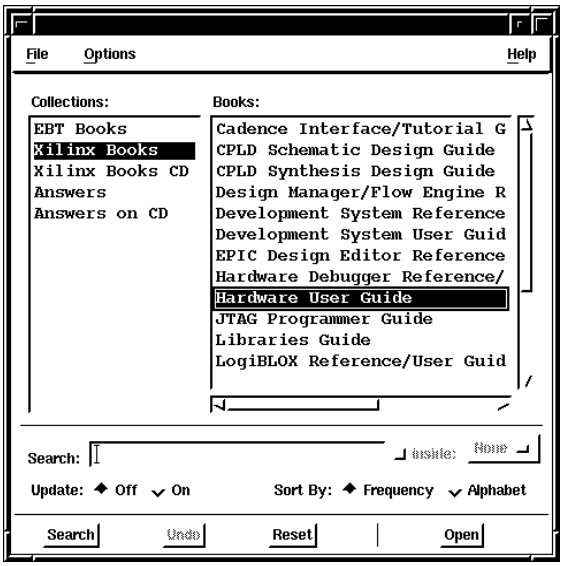

Figure 38: Dynatext Browser window.

APPENDIX

This section contains a printout of Chapter 1 from the Hardware User Guide available through the Xilinx on-line documentation.