Introduction to Synopsys and VHDL on Solaris

© Naveen Michaud-Agrawal
for Dr. Pakzad’s CSE 331 Honor class
September 25, 2000

1 Introduction

VHDL is an acronym which stands for VHSIC Hardware Description Language. VHSIC is yet another acronym which stands for Very High Speed Integrated Circuits. The key idea behind VHDL is the abstraction of a design from its implementation. The power of VHDL lies in its ability to describe a testable design without making any assumptions about the underlying hardware used to implement it. This is a very powerful paradigm.

One of the key features of VHDL is its versatility. It can be used for documentation, verification, and synthesis of large digital designs. A single design can theoretically achieve all three of these goals, thus saving a lot of effort. Although VHDL is a language, VHDL designs are unlike programs written in C, Pascal or other structural languages. The thought processes that go into writing programs in these languages is inherently procedural, which reflects the serial nature of most modern computers. VHDL programs, on the other hand, attempt to mimic the behavior of a physical system such as a digital circuit.

Digital systems are fundamentally about discrete signals. When the value of a signal changes, we say an event has taken place on that signal. Signals are connected to other signals through gates. When one signal changes, the change is propagated to all signals connected to the signal that changed. In an idealized world this transition occurs instantaneously, but real world components take a certain finite amount of time to switch states (due to inertia). This period of time is called the propagation delay, and is a function of the physical properties of the gates and wires.

Another property of digital systems is their concurrency. Consider a signal connected to two gates. When the value of the signal changes, the outputs of both gates change depending on the value of the signal. If both gates have the same propagation delay then they will produce events on their outputs simultaneously. These new events will propagate through the rest of the circuit and initiate new computations that occur concurrently.

Most digital systems are characterized by complex interactions that are difficult to explicitly compute. Therefore we must accurately model them to ensure that they satisfy our design goals. VHDL is able to model systems at many levels of abstraction, from the transistor level to the computing system level.

A design consists of two components, the interface of the design, meaning its inputs and outputs, and the behavior that specifies how those inputs are mapped to outputs. Typically there are many ways to describe the behavior of a design, but generally they fall into two design classes. A description consisting of the components of a design and their interconnections is termed a structural description. The other type of description, is actually broken into two subclasses: data flow descriptions and behavioral descriptions.
A data-flow description considers the type of low-level processing a design does to transform input signals into output signal, while a behavioral description uses a higher level of abstraction to model a design.

The scheme that the VHDL standard proposes to simulate circuits is called a *discrete event time model*. The simulator maintains a list of all events timestamped with the time they are to occur. This list is ordered according to increasing timestamp value. The simulator proceeds by processing all events that occur at the current time and then incrementing the clock. The processing of these events generates events on other signals which will occur in the future. As events are generated they are timestamped and placed on the list in the appropriate location. When all events at the current time have been processed, the simulator increments the simulator clock to the time of the next event.

This tutorial does not cover behavioral description. Sometimes signal assignment statements are not enough to capture the essence of a design. In this case VHDL provides a special construct, called a process, that is similar to a program written in a sequential language, i.e., you can have variables, loops, and other sequential constructs. The whole process is treated as a single signal assignment statement and is scheduled the same as other CSAs. This means that in simulation time the whole process finishes in zero time. For more information, refer to any book on VHDL.

VHDL is a standard (VHDL-1076) developed by IEEE (Institute of Electrical and Electronics Engineers). The language has been through a few revisions, and you will come across this in the VHDL community. Currently, the most widely used version is the 1987 (std 1076-1987) version, sometimes referred to as VHDL'87, but also just VHDL. However, there is a newer revision of the language referred to as VHDL'93. VHDL'93 (adopted in 1994 of course) is fairly new and is still in the process of replacing VHDL'87.

The remainder of this tutorial discusses the use of VHDL'93 for hardware description and verification using industry standard tools from Synopsys.

## 2 Installation

First you must prepare your account for Synopsys by editing `~/.cshrc.cat`. Look for the following lines (a line starting with "##" is a comment):

```bash
## Synopsys CAD Tools
## if (-f $confdir/cshrc.synopsys) source $confdir/cshrc.synopsys
```

Uncomment the second line, which results in the following:

```bash
## Synopsys CAD Tools
if (-f $confdir/cshrc.synopsys) source $confdir/cshrc.synopsys
```

Save the file and quit your text editor. Now type in `source ~/.cshrc.cat` for the changes to take effect. This change will allow you to use the Synopsys analyzer and simulator tools.

The next step is to create a project directory where you will store all your vhdl files. Synopsys requires that preference files be kept in the same directory as the vhdl files. Preferences are stored in two files: `~/.synopsys_vcc.setup` and `~/.synopsys_dc.setup`. You can find a sample `synopsys_vcc.setup` and `synopsys_dc.setup` in `~cg331/`. Copy both files to your directory:

```bash
% mkdir [your vhdl directory]
% cd [your vhdl directory]
% cp ~cg331/.synopsys_vcc.setup ~cg331/.synopsys_dc.setup.
```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>cin</th>
<th>sum</th>
<th>cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(a)

Figure 1: Truth table and circuit of full adder

You will only need to edit two lines in `.synopsys.vcc.setup`. Open `.synopsys.vcc.setup` with a text editor and look for the line starting with `TIMEBASE`. Change it so that it looks like:

\[
\text{TIMEBASE} = \text{NS}
\]

This sets the base time unit of the simulator to nanoseconds. Now look for the line starting with `WAVEFORM`. Change it so that it looks like this:

\[
\text{WAVEFORM} = \text{WIF+WAVES}
\]

This will allow you to see waveforms of your simulations (a topic we will cover later on).

3 VHDL Explained

In the course of this tutorial we will build a full adder and use it to build a simple 4-bit adder. We will use a design of a full-adder (see Listing 1 on page 4) to demonstrate the data-flow paradigm and a 4-bit adder (see Listing 2 on page 6) to demonstrate the structural design method. These two methods are not necessarily exclusive, and you will often find it useful to combine them.

3.1 Data Flow Design

To start, let us consider a simple full adder. The truth table and corresponding circuit diagram are shown in Figure 1. Listing 1 describes the VHDL model of the full adder. The first three lines are comments. A comment can occur at the end of a line or on its own lines, and are specified using two dashes (`--`). The VHDL analyzer ignores everything after the comment symbol.

VHDL uses the concept of libraries and packages of standard components. The main one that is almost always used is the IEEE standard library. Within this library you will want to use the `std_logic_1164` package, which defines the standard signal types and their values. VHDL accesses these through the use of the `library` and `use` keywords.

The first thing to notice is the `entity` statement. Much like C, VHDL requires a declaration of the interface of a design before it can be described. Notice that the entity is named `full_adder`. One thing to keep in mind is that VHDL is case-insensitive.
-- ************************************************************
-- A full adder
-- ************************************************************
library IEEE;
use IEEE.std_logic_1164.all;

entity full_adder is
port(a, b, cin : in std_logic;
    sum, cout : out std_logic);
end full_adder;

architecture behavior of full_adder is
signal s0, s1, s2 : std_logic;
begin
  s0 <= a xor b after 2 ns;
  s1 <= a and b after 1 ns;
  s2 <= cin and s0 after 1 ns;
  sum <= s0 xor cin after 2 ns;
  cout <= s2 or s1 after 1 ns;
end behavior;

Listing 1: VHDL code for full adder

The inputs and outputs of an entity are called ports. Ports are special programming objects, namely signals. Like variables in a conventional language, each port must be a signal of a specified type. In this case each port is declared to be of type std_logic. The IEEE 1164 standard defines std_logic as a standard signal that can assume one of nine values, listed in Table 1. You can think of a std_logic as a special bit value.

<table>
<thead>
<tr>
<th>Value</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>Uninitialized</td>
</tr>
<tr>
<td>X</td>
<td>Forcing Unknown</td>
</tr>
<tr>
<td>0</td>
<td>Forcing 0</td>
</tr>
<tr>
<td>1</td>
<td>Forcing 1</td>
</tr>
<tr>
<td>Z</td>
<td>High Impedance</td>
</tr>
<tr>
<td>W</td>
<td>Weak Unknown</td>
</tr>
<tr>
<td>L</td>
<td>Weak 0</td>
</tr>
<tr>
<td>H</td>
<td>Weak 1</td>
</tr>
<tr>
<td>-</td>
<td>Don’t Care</td>
</tr>
</tbody>
</table>

Table 1: IEEE 1164 signal values

Another useful signal type is std_logic_vector. An std_logic_vector is a signal type comprised of an array of signals, each of type std_logic. The std_logic and std_logic_vector types are the two most common types. In general, a signal may be one of several other VHDL data types, such as integer or real, but these will not be considered in this tutorial. We will see a case of the std_logic_vector later on in the example of the 4-bit adder.

The signals appearing in a port declaration may be input signals, output signals, or bidirectional signals. This is referred to as the mode of the signal. Input signals are designated by in, output by out, and bidirectional signals as inout. Every port in the entity declaration must have a mode specified.

It is now necessary to describe the internal behavior of the full adder. The VHDL construct that allows us to do this is the architecture construct. In Line 13 in Listing 1 the architecture statement describes a module named behavior which will describe the behavior of the design entity full_adder. The behavioral description provided in the architecture can be at one of many levels of abstraction. These differ in their level of detail, description of events and degree of detail. Thus the description of a design entity takes the form of an entity-architecture pair, although an entity may be described by more than one architecture.

The first thing after the architecture statement is the declaration of three signals of type std_logic
3.2 Structural Design

(line 14). These signals are internal to the design and are used to connect input signals to output signals. It is possible to map the input signals directly to the output signals but usually it makes for a cleaner design to separate the output for each gate into a separate signal. Figure 1 illustrates the use of these three internal signals.

Between the begin and end statements of the architecture block are five concurrent signal assignment (CSA) statements. These statements describe the way the signal on the left-hand side changes when one of the signals on the right hand-side changes. The after keyword specifies the delay between the change in the input to CSA and the corresponding change in the output. For example, if a changes from 0 to 1, then statements on lines 16 and 17 will be executed. If s0 or s1 change as a result, this transition will occur 2 nanoseconds after the current time. Strictly speaking, the use of the after keyword is not required in a design, but it is useful to accurately simulate a real digital system. If the time delay is omitted then all CSAs will occur instantaneously. For example, when one of the inputs to the full adder changes, the outputs will change at the same time, and since their is no propagation delay, the sum and carry-out will be computed instantaneously.

By way of analogy, the entity construct is like the pin description within a data book, while the architecture construct is like a schematic for the design.

3.2 Structural Design

Next we will look at the design of a 4-bit adder. A 4-bit adder can be built using 4 full adders chained in succession (see Figure 2). One of the advantages of VHDL is the ability to define building blocks that can be used to build more complicated designs. The VHDL code for the 4-bit adder is shown in Listing 2.

Again, the first thing declared is an entity construct. All designs, whether structural or behavioral, are composed of entity-architecture pairs. The in ports a and b and the out port sum are defined as std_logic_vectors. The declaration std_logic_vector( 3 downto 0) defines a vector of 4 bits with the most significant bit in the leftmost position.

The first thing to notice in the architecture construct is the definition of three signals c0, c1, and c2 of type std_logic (line 16). These three signals are internal signals and are not seen outside of the circuit. Later we will see that these signals are used to connect the carryout from one full adder to the carry in of another.
-- ******************************************************************************
--  A 4-bit adder
-- ******************************************************************************

library IEEE;
use IEEE.std_logic_1164.all;

entity adder_4bit is
  port(a, b: in std_logic_vector(3 downto 0);
      sum: out std_logic_vector(3 downto 0);
      cin: in std_logic;
      cout: out std_logic);
end adder_4bit;

architecture structure of adder_4bit is
  signal c0, c1, c2: std_logic;

  component full_adder
    port(a, b, cin: in std_logic; sum, cout: out std_logic);
  end component;

  for all: full_adder use entity work.full_adder(behavior);

begin
  fa0: full_adder port map(a(0), b(0), cin, sum(0), c0);
  fa1: full_adder port map(a(1), b(1), c0, sum(1), c1);
  fa2: full_adder port map(a(2), b(2), c1, sum(2), c2);
  fa3: full_adder port map(a(3), b(3), c2, sum(3), cout);
end structure;

Listing 2: VHDL code for 4-bit adder
The next block of code (lines 18-22) forms a component declaration. A component declaration describes the interface of a component so that it can be used later on in the description of a design. Once a component declaration is defined, it must be bound to a particular architecture. This is akin to selecting a particular implementation of a chip interface when physically building a circuit. In this case all full_adder components within the 4-bit adder will use the behavior architecture of the full adder (line 22). The work.full_adder(behavior) phrase specifies the behavior architecture of the full_adder entity found in library work. Remember that VHDL uses libraries to group entities. All entities that aren’t explicitly grouped into a library (a topic we will not cover) are bound to the library work.

After the component declarations we see the description of the architecture. Lines 25-28 describe component instantiations, which are actual instances of the components used within the design. The purpose of the component instantiation statement is to describe the connection between the signals of the design and the interface of the component. The first word is a label for the component and must be unique within the entity. The next part specifies the component used (in this case full_adder) followed by the port map clause. The port map clause specifies which signals of the design are connected to the ports of the component in the order that they are described in the component declaration. As an example, we will take the component instantiation on line 25 and determine its meaning. This component is a full adder and is labeled fa0 (for full adder 0). The port map clause specifies that the least significant bit of the input signal a (designated as a(0)) of the 4 bit adder is interfaced with the a input port of the full adder. Likewise, input b(0) is mapped to input port b, cin to cin, sum(0) to sum and c0 to the cout of the full-adder. Three more of these components are defined, with the carry out of one full adder becoming the carry in of the next one in sequence. Figure 2 shows how this would look schematically. The output of these four full-adders is mapped onto the circuit’s 4-bit output.

4 Analyzing VHDL

After a design has been written in VHDL, it must be put into a form that a simulator can understand. This process is called analyzing, a term synonymous with compiling. To analyze the code, type “vhdlan [filename]” at the command prompt. If you have typed the code correctly, there should be no errors. The most common type of error is a syntax error. One of the most annoying things about Synopsys’s analyzer is that almost every error is considered a syntax error. One common error occurs when a semicolon has been omitted, and usually the rest of the error messages following the syntax error occur because the analyzer is confused. Other lexical errors such mispellings are also signaled as syntax errors. Another syntax error occurs when the analyzer cannot find the analyzed version of a component for the current design. This error can be avoided by analyzing lower level designs before analyzing the designs that use them as components.

vhdlan outputs several files for each entity declaration and each architectural description. If you do a directory listing after analyzing, you will notice a .sim file for every entity (and a .mnf if the architecture contains any CSA’s) and likewise for each architecture. The entity files are prefixed in capital letters by the name of the entity and the architecture files are prefixed by the entity name followed by the name of that particular architecture.

5 Simulating a VHDL Design

Synopsys comes with a VHDL simulator named vhdlsim. To simulate a particular design, type “vhdlsim” followed by the name of the architecture file you wish to simulate. If you have more than one architecture
specified for that entity, vhdlsim defaults to the last one defined in the VHDL file. To use a different architecture than the default specify the full name of the architecture (entity_architecture; the argument is case-insensitive). For example, to test the full adder described above type “vhdlsim full adder” (you don’t need to specify the architecture since only one has been defined). Suppose that there are two architectures describing full adder, arch_a and arch_b. To simulate arch_a you would type “vhdlsim full adder_arch_a”.

vhdlsim is a command-line simulator. The architecture is organized in a directory-like hierarchy, with the entity you are simulating sitting at the root of the tree. To get a list of all the commands, type “help”. vhdlsim also has help information for each command, which can be accessed by typing in “help <command>”. Entities within the hierarchy are referred to by their full pathnames. For example, the 4-bit adder above contains 4 full-adders, labeled fa0 to fa3. To access the value of the input signal cin of the third full-adder the command is “ls -v adder_4bit/fa2/c”.

To traverse the hierarchy use “cd <entity>”. The first thing you will want to do is to “cd” into your design. “ls” lists all the ports, signals, components, component instantiations, and CSAs within the current entity. Use “ls -t” to list the types of these objects, and “ls -v” to list the values of all the objects. Another useful command is “comm”. “ls” can also take wildcards. For example, “ls -v *signal” lists the values of all the signals at the current level of the hierarchy (“signal” specifies the type, in this case a signal).

Signals can be assigned values with the “assign <value> <signal>” command. When assigning to a std_logic signal, the value must be surrounded by single quotes. When the signal is of type std_logic_vector, the assigned value must be in double quotes. It is also possible to specify the value in hexadecimal format if the size of the vector is a multiple of 4. This is done by prefixing the value with an x. For example, supposed datain is defined to be of type std_logic_vector(15 downto 0). The command “assign x’FFFFFF’, datain” will assign datain the value of 0000000000000000.

The command “run <timevalue>” is used to progress through the simulation. <timevalue> specifies the amount of time to progress; if it is omitted the simulator runs until the effects of all concurrent signal events have been propagated through the design and there are no more events to process.

Sometimes you would like trace the waveforms of several signals simultaneously. vhdlsim allows this with the “trace <signal 1> <signal 2> ...” command. You can list as many signals as you want, but remember not to put commas between them. The “trace” command will bring up a viewer from which you can see the waveforms being traced as the simulation progresses.

Frequently used groups of commands can also be stored into a macro. This is done with the “comm” command. Type “comm” and press enter, and then type in all the commands you would like to store within the macro. Type “end” when you are finished to indicate to vhdlsim that you are done defining the macro. You can also store a sequence of commands in an external file and load them into the simulator when it is started using “vhdlsim -i [commandfile] [entity]”. These commands are run before the interactive shell starts. This is useful to set up preliminary values for the signals before the simulation runs. Finally, to quit the simulator type “quit” at the prompt. To reset the simulation you must quit and then rerun the simulator.

As an example, we will go through the simulation of the full adder described earlier. First type “vhdlsim full adder” to start the simulator. Some information about the simulator and a copyright notice will scroll by, and then you will be presented with a prompt (marked with a “#”). First type in “ls -t” (remember that -t specifies a type listing). You should get the following output:

# ls -t
FULL_ADDER COMPONENT INSTANTIATION STATEMENT
STANDARD PACKAGE
ATTRIBUTES PACKAGE
STD_LOGIC_1164 PACKAGE
KERNE PROCESS STATEMENT

The only important object is FULLadder. The other objects are part of the simulator's runtime. Now type in “cd full_adder” followed by “ls -t”.

# cd full_adder
# ls -t
A IN PORT
type = STD_LOGIC
B IN PORT
type = STD_LOGIC
CIN IN PORT
type = STD_LOGIC
SUM OUT PORT
type = STD_LOGIC
COUT OUT PORT
type = STD_LOGIC
_P0 PROCESS STATEMENT
_P1 PROCESS STATEMENT
_P2 PROCESS STATEMENT
_P3 PROCESS STATEMENT
_P4 PROCESS STATEMENT
S0 SIGNAL
type = STD_LOGIC
S1 SIGNAL
type = STD_LOGIC
S2 SIGNAL
type = STD_LOGIC

Here you can see all the objects that comprise the full adder and their types. P0, P1, P2, P3, and P4 are just representations of the five concurrent signal assignment statements in the architecture declaration. Now type in “ls -v” to see the values of the signals:

# ls -v
A ‘U’
B ‘U’
CIN ‘U’
SUM ‘U’
COUT ‘U’
_P0 (no value)
_P1 (no value)
_P2 (no value)
_P3 (no value)
_P4 (no value)
As you can see, all the signals have a value of ‘U’, which means they are uninitialized. Before we assign values to them, however, we will set up a trace of the waveforms of signals \(a, b, \text{cin}, \text{sum}, \text{cout}\). After typing in “trace a b cin sum cout” a window will appear (Figure 3). Go back to the simulator console window and type “run 10”. You will see waveforms for all the signals, however they are compressed (Figure 4). The \(Z^+\) and \(Z^-\) buttons cause the waveform area to zoom in and out. Notice that the current simulator time is 10 (the units in this case are nanoseconds, although this isn’t shown) and all the signals have a value of ‘U’. Now set the value of \(a\) to 1 with “assign ’1’ a” and do an “ls -v”:

```vhd
# assign ’1’ a
# ls -v
A ’1’
B ’U’
CIN ’U’
SUM ’U’
COUT ’U’
.P0 (no value)
.P1 (no value)
.P2 (no value)
.P3 (no value)
.P4 (no value)
S0 ’U’
S1 ’U’
S2 ’U’
```

Run the simulator for another 10 nanoseconds (“run 10”) and look at the waveform window. Notice that none of the other values have changed. This is because VHDL calculates any logic operation where one
Figure 4: Showing compression of waveforms in waveform viewer

(or both) of the values are uninitialized as a result that is uninitialized. This means that all input signals must have a value of either 0 or 1. Set the values of a, b, and cin to 0 and run the simulator for 10 nanoseconds (ns). Notice that values of sum and cout change at different times because they experience different propagation delays. Now set a to 1 and run the simulation. “ls -v” shows:

```
# assign '1' a
# run 10
40 NS
# ls -v
A     '1'
B     '0'
CIN   '0'
SUM   '1'
COUT  '0'
_P0   (no value)
_P1   (no value)
_P2   (no value)
_P3   (no value)
_P4   (no value)
S0    '1'
S1    '0'
S2    '0'
```

The sum has changed to reflect the full adder's computation. Figure 5 shows the waveforms for the sequence of assignments corresponding to the truth table for the full-adder (Figure 1(a)). As an exercise simulate the design for the 4-bit adder.
6 Creating Schematics

Synopsys also includes a tool named *Design Analyzer* to create schematics from VHDL descriptions. Again we will use the full-adder to demonstrate this. Here are the steps necessary to create and print a schematic for the full-adder design:

1. *Design Analyzer* requires its own directory to store all the files it generates. Create a directory named `work` in your project directory. If you followed the instructions in the beginning of the packet and copied `.synopsys_dc.setup` into your project directory then Design Analyzer is all set to run.

2. Run *Design Analyzer* by typing “`design_analyzer`” on the command line and pressing enter. *Design Analyzer*’s main window should appear (see Figure 6).

3. Open `full_adder.vhd` by clicking on **File->Analyze...** in the menu bar. Select `full_adder.vhd` from the directory listing and click OK. A window should pop up and output some information about the analyzing process. If there are no errors click the Cancel button (*Design Analyzer* is weird in that some of the windows are closed using the Cancel button, although this doesn’t undo what you just
Figure 7: Main window showing symbolic representation of full-adder

Figure 8: View Style window
did). Now open the File->Elaborate... menu. In the first list box select DEFAULT to access the default library (which is library WORK in this case). You should see full_adder(behavior) appear in the next list box. Select that and click OK. Again a message window should appear; just press the Cancel button when it is done (the last line should read “design_analyzer>”). The main window now has a small AND symbol representing the full-adder design (Figure 7).

4. To create easy to read schematics several view parameters must be changed. In the View->Style... menu (Figure 8) modify the following settings (make sure to press the Apply button after you change each setting):

- bus_ripper_name_layer: Visibility Off.
- cell_name_layer: Visibility On.
- cell_ref_name_layer: Visibility Off.
- net_name_layer: Visibility On.

![Figure 9: Main window showing abstraction of design](image)

Press the Cancel button when you are done. Unfortunately, Design Analyzer does not store these changes when you quit, so this must be done everytime you rerun Design Analyzer.

5. Now doubleclick the icon for the full-adder. The view should change to an abstraction of the design showing only the inputs and outputs, and the bottom right hand corner of the window should say Symbol View (Figure 9). You can also access this view by selecting the icon for the full-adder and pressing the descend-hierarchy button.

6. To create a schematic press the schematic button. The view should change to a schematic (Figure 10). You can zoom in by using the corresponding menu options in the View menu and scrolling around the view.

7. To print the schematic open the printing window from the File->Plot... menu and press the OK button.
When you first create the schematic Design Analyzer uses 2 input gates. If you have a complicated design the schematic will be very large and hard to understand. To minimize the design you must set several Optimization Directives. Click on Attributes->Optimization Directives->Design... to open the Design Attributes window. There are four sections each separated by a separator. Ignore the first two sections. In the third section make sure Flatten Logic is selected. Select High for Flatten Effort, Multiple Output for Flatten Minimize, and Apply Strategy for Flatten Phase. In the last section make sure Structure Logic is deselected (see Figure 11) and press the Apply button. Finally, select the Tools->Design Optimization... menu item. None of the settings need to be changed so just press OK. In the case of the full-adder Design Analyzer combines several of the gates and uses NAND and NOR gates since they are easier to build.
Once you created the schematic for the full adder repeat the above steps for the 4-bit adder. In your own projects, remember to load lower level components before the designs that use them (similar to vhdlan), otherwise Design Analyzer will signal an error. For more help with using Design Analyzer, access the online help with Help->On-Line Documentation.
A Listings

List of Figures

1  Truth table and circuit of full adder .................................................. 3
2  4-bit adder circuit .................................................................................. 5
3  Synopsys waveform viewer ................................................................. 10
4  Showing compression of waveforms in waveform viewer .................. 11
5  Waveform for full adder simulation ....................................................... 12
6  Main window for Design Analyzer .......................................................... 12
7  Main window showing symbolic representation of full-adder ............. 13
8  View Style window ............................................................................... 13
9  Main window showing abstraction of design ......................................... 14
10 Main window showing schematic of full-adder ....................................... 15
11 Design Attributes window .................................................................... 15

List of Tables

1  IEEE 1164 signal values ......................................................................... 4

List of Code Listings

1  VHDL code for full adder .......................................................................... 4
2  VHDL code for 4-bit adder ...................................................................... 6