Module 5: Introduction to VHDL

Revision: August 30, 2007

Produced in cooperation with DIGILENT®
www.digilentinc.com

Overview

Since the first widespread use of CAD tools in the early 1970's, circuit designers have used both picture-based schematic tools and text-based netlist tools. Schematic tools dominated the CAD market through the mid-1990's because using a graphics editor to build a structural picture of a circuit was easy compared to typing a detailed, error-free netlist. But early graphics-based tools came with a heavy price – expensive graphics-capable workstations were required to run them, and designs could not be transferred between computers or between CAD tools. Early text-based tools, which essentially just allowed designers to type netlists directly, gained momentum because the tools weren't tied to high-end computers.

As progress in IC fabrication technologies made it possible to place more and more transistors on a chip, it became apparent that schematic methods were not scaling very well to the more complex design environments. A single designer could specify the behavior of a circuit that required several thousand logic gates, but it took several layout engineers many weeks or months to transfer that behavior to patterns of transistors. As designs increased in complexity, more engineers were employed on larger teams, and a much larger volume of detailed technical data had to be shared between workers.

Technology advances created new bottlenecks – it was proving difficult to keep large numbers of designers and layout engineers all up to date with precise specifications in complex and evolving design environments. In response, the US Department of Defense began a program to develop a method by which designers could communicate highly specific technical data. In 1981, the DOD brought together a consortium of leading technical companies, and asked them to create a new "language" that could be used to precisely specify complex, high-speed integrated circuits. The language was to have a wide range of descriptive capability, so that detailed behaviors of any digital circuit could be specified. This work resulted in the advent of VHDL, an acronym for "Very-high-speed-integrated-circuit Hardware Description Language". This module presents several of the basic concepts involved in using VHDL as a design tool for digital circuits. In subsequent modules, further discussions of the VHDL language will keep pace with circuit descriptions.

Before beginning this module, you should…

- Be familiar with the structure of logic circuits
- Know how to use the WebPack schematic tools to enter and simulate circuits
- Know how to download circuits to the Digilent circuit board.
- Understand logic systems and minimization techniques

After completing this module, you should…

- Be able to enter a VHDL description of a combinational logic circuit
- Be able to synthesize, simulate, and download a VHDL-based circuit
- Understand the role of VHDL and circuit synthesizers, and the difference between structural and behavioral designs

This module requires:

- A Windows PC
- The Xilinx ISE/WebPack software
- A Digilent circuit board

Background

VHDL was introduced as a means to provide a detailed design specification of a digital circuit, with little thought given to how a circuit might be implemented based on that specification (the assumption was the requirements in the source file would be captured as a schematic by a skilled engineer). At the time, the creation of a design specification, although involved, was almost trivial in comparison to the amount of work required to translate the specification to a schematic-based structural description needed to fabricate a device. Over several years, it became clear that a computer program could be written to automatically translate a VHDL behavioral specification to a structural circuit, and a new class of computer programs called synthesizers began appearing. A synthesizer produces a low-level, structural description of a circuit based on its HDL description. This automated behavioral-to-structural translation of a circuit definition greatly reduced the amount of human effort required to produce a circuit, and the VHDL language matured from a specification language to a design language.

The use of HDLs and synthesizers has revolutionized the way in which digital engineers work, and it is important to keep in mind how rapidly this change has come about. In 1990, very few new designs were started using HDLs (the vast majority were schematic based). By the mid 1990's, roughly half of all new designs were using HDLs, and today, all but the most trivial designs use HDL methods. Such rapid change demonstrates that engineers overwhelmingly recognize the advantages of using HDLs. But such rapid change also means that tools, methods, and technologies are still evolving, and that CAD tools are continuing to be developed and improved.

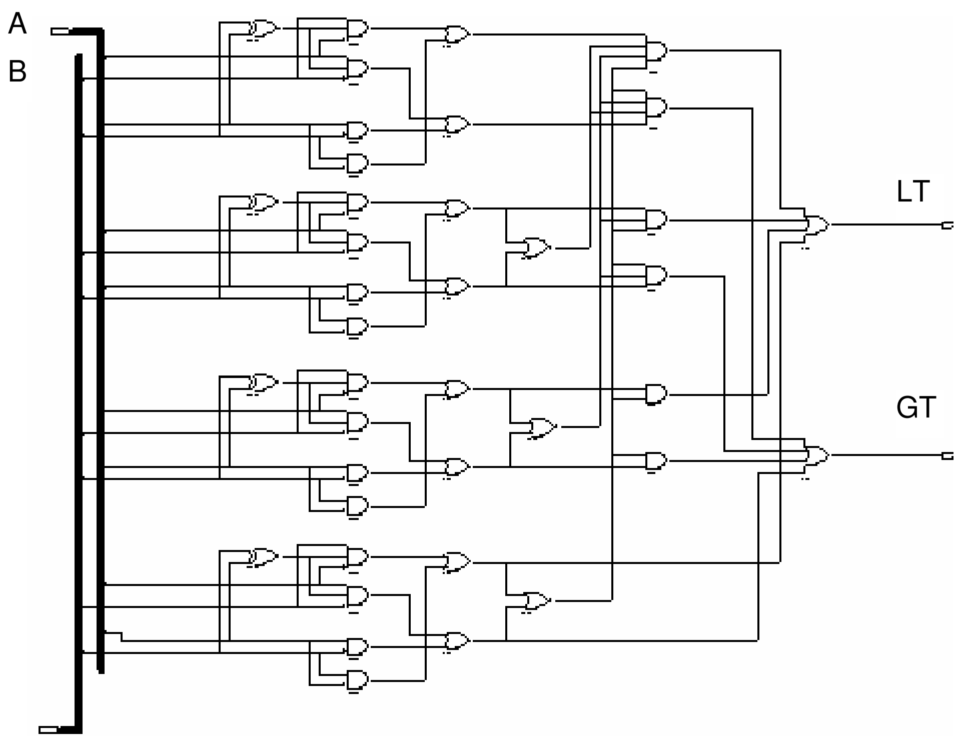
Digital design CAD tools can be placed in two major categories – the "front-end" tools that allow a design to be captured and simulated, and "back-end" tools that synthesize a design, map it to a particular technology, and analyze its performance (thus, front-end tools work mostly with virtual circuits, and back-end tools work mostly with physical circuits). Several companies produce CAD tools, with some focusing on front-end tools, some on back-end tools, and some on both. Two major HDLs have emerged – one developed by and for private industry (called Verilog), and the other fueled by the government and specified by the IEEE (VHDL). Both are similar in appearance and application, and both have their relative advantages. We will use VHDL, because a greater number of educational resources have been developed for VHDL than for Verilog. It should be noted that after learning one of the two languages, the other could be adopted quickly.

HDLs have allowed design engineers to increase their productivity many fold in just a few years. It is fair to say that a well-equipped engineer today is as productive as a small team of engineers just a few years ago. Further, hardware specification is now within the reach of a wider range of engineers; no longer is it the domain of only a few with highly specialized training and experience. But to support this increased level of productivity, engineers must master a new set of design skills: they must be able to craft behavioral circuit definitions that provably meet design requirements; they must understand synthesis and other CAD tool processes so that results can be critically examined and interpreted; and they must be able to model external interfaces to the design so that it can be rigorously tested and verified. The extra degree of abstraction that HDL allows brings many new sources of potential errors, and designers must be able to recognize and address such errors when they occur.

Structural vs. Behavioral design

A behavioral circuit design is a description of how a circuit's outputs are to behave when its inputs are driven by logic values over time. A purely behavioral description provides no information to indicate how a circuit might be constructed – that information must inferred from the definition through application of several pre-designed rules. As an example, consider the following behavioral definition written in proper VHDL syntax: *GT <= '1' if A > B else '0'*. The GT ("greater than") output could be formed by a processor circuit doing a comparison under the control of software, or by the "borrow out" of a hardware subtractor circuit, or by a custom-designed logic circuit. Any of these implementation methods would meet the behavioral requirements contained in the VHDL statement.

A structural circuit definition is essentially a plan, recipe, or blueprint of how a circuit is to be constructed, and it is required before a circuit can be constructed. In its most detailed and basic form, a structural definition provides no information to indicate how a circuit might behave. Consider the structural circuit definition (schematic) shown. To discover the high-level behavior of this circuit (assuming no prior knowledge of the design), a time consuming and detailed analysis would need to be performed. But its behavior can be stated rather succinctly: assert LT if the 4-bit input number A is less than B, and GT if A is greater than B.

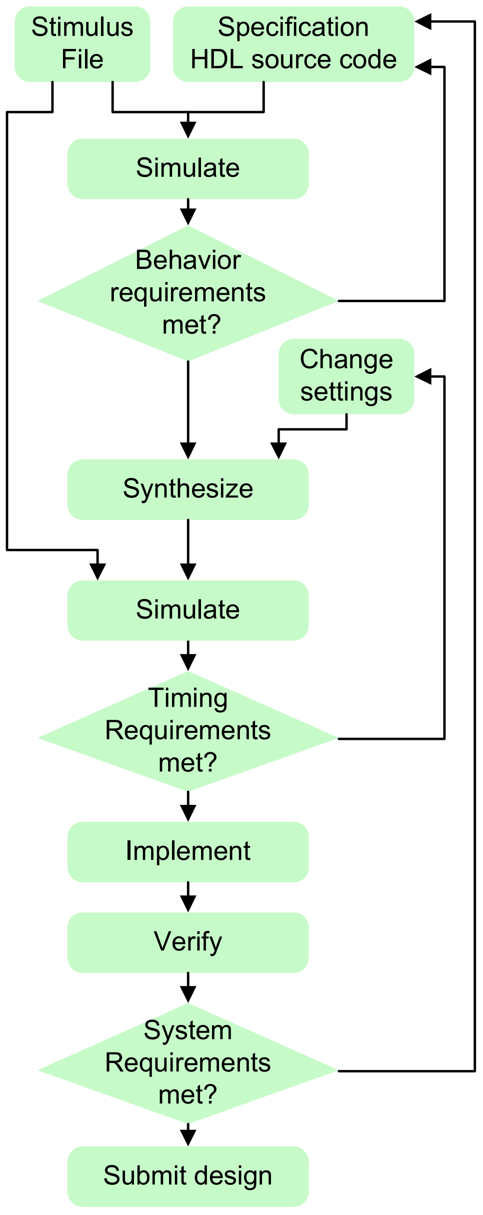


Schematic for a 4-bit magnitude comparator circuit

HDL source files can be written to define circuits using behavioral methods or structural methods (or more commonly, a mixture of the two). In any case, an HDL source file must be synthesized into a structural description before a circuit can be implemented. When a behavioral circuit is synthesized, the synthesizer must search through a large collection of template circuits, and apply a large collection of rules to try to create a structural circuit that matches the behavioral description. The synthesis process can result in any one of several alternative circuits being created due to the variability inherent in generating rule-based solutions. But when a structural description is synthesized, the synthesizer's job is a relatively straightforward, involving far fewer rules and inferences. A post-synthesis structural circuit will closely resemble the original structural definition. For this reason, many designers prefer to use a "mostly structural" approach, even though this approach does not capitalize on one of the major benefits of using VHDL (i.e., the ability to quickly and easily create behavioral designs).

In general, it is far easier and less time consuming to define a given circuit using behavioral methods than to define the same circuit using structural methods. Behavioral descriptions allow engineers to focus on high-level design considerations, and not on the details of circuit implementation. But while behavioral methods allow engineers to design more complex systems more quickly, they don't allow engineers to control the structure of their final circuit. Synthesizers must use rules that are applicable to a wide range of circuits, and they cannot be optimized for a particular circuit. In some situations, engineers must have greater control over the final structure of their circuits. In these cases, structural methods are more appropriate. Often, engineers might start a new design with a behavioral description so that they can readily study the circuit and possible alternatives. Then, once a particular design has been chosen, it can be recoded in structural form so that the synthesis process becomes more predictable. Structural descriptions read rather like a netlist, and although they are more difficult to create, it is straightforward to sketch circuit from a structural HDL source file.

Instead of using a graphics interface to add gates and wires to a schematic, HDLs editors use a text editor to add structural or behavioral descriptions to a text file. Behavioral descriptions describe the conditions required for a given signal to take on a new value. For example, the VHDL statement Y <= (A and B) or (not A and B and C) or (not A and not C), read as "Y gets assigned AB + A'BC + A'C'", describes only how Y is to behave, and not how a circuit that performs the operation is to be built. Structural descriptions use components interconnected by signal names to create a netlist (see problem 1 above for an example). Whether a circuit is coded structurally or behaviorally, it must be synthesized before it can be implemented, and the synthesizer automatically minimizes all logic equations.



HDL design flow

Synthesis and Simulation

A VHDL design can be simulated to check its behavior, and/or synthesized so that it can be implemented. These two functions, simulation and synthesis, are really separate functions that do not need to be related. In a typical flow, a new design would be simulated, then synthesized, and then simulated again after synthesis to ensure the synthesizer did not introduce any errors. But it is possible to eliminate either (or both) simulation steps altogether, and proceed directly to synthesis. It is also possible to simulate a new design many, many times prior to synthesis so that various design alternatives can be investigated. In either case, a typical first step is to use a CAD tool called an analyzer to check the VHDL source for any grammar or syntax errors. Code that analyzes successfully can be routed to the simulator or synthesizer.

Although the simulator used in HDL environments is similar to the simulator used in schematic environments, there are a few key differences. One difference is that a schematic must be rendered into a netlist before it can be simulated, but a VHDL source file can be simulated directly (or, restated, a VHDL source file can be simulated before synthesis). This difference results from the fact that in a schematic environment, symbols added to a circuit are really just graphical placeholders for simulation routines. In an HDL environment, no such simulation routines exist – the user must define each circuit's behavior in proper VHDL syntax, and the simulator directly uses these definitions. Another difference is that VHDL environments are more tightly integrated with their simulation environments, and several VHDL language features exist solely for use in circuit simulations (more on this later).

A VHDL circuit description must be synthesized before it can be implemented in a given device or technology. A typical synthesis process transforms a behavioral description to basic logic constructs such as AND, OR and NOT operations (or perhaps NAND and NOR operations), and these basic operations are then mapped to the targeted implementation technology. For example, a given design might be mapped to a programmable device like an FPGA, or it might be mapped to a fully custom design process at a semiconductor foundry. The desired target technology is specified during the synthesis step, which means that the same VHDL source file can be used to create a prototype design that will be downloaded to an FPGA, or it can be used to create a custom chip. This ability to use the same source file to implement a design in vastly different technologies is a key strength to using HDL design methods.

The design flow on the right shows the steps involved with HDL based designs. Note that two simulation steps occur – one just after the HDL code is entered, and the other after the design has been synthesized. The first simulation step allows designers to quickly check the logical behavior of their design, before any thought or effort is applied to implementing a physical circuit. This early simulation allows several architectural alternatives to be compared and contrasted early in the design cycle, before decisions are “locked in” to a particular hardware solution. The second simulation step allows designers to verify the design still works after it has been synthesized and mapped to a given hardware device.

A VHDL source file contains no information to direct how a given circuit might be implemented. Most designs must meet stringent timing requirements, or power consumption limits, or size specifications. During the synthesis operation, the designer can constrain the synthesizer to optimize the process for power consumption, for area, or for operating speed. The post-synthesis simulation allows a designer to check that the synthesis process created a physical circuit that meets the original design specifications. If the specifications are not met, the designer can re-run the synthesis process with new constraints.

In the recent past, the majority of the engineering effort required for a new design was applied to transferring high-level specifications into low-level structural descriptions – the exact function that synthesizers perform today. Although this very detailed process required much effort, it also allowed designers to gain a very detailed understanding of the actual, physical circuit. Using synthesizers to perform these tasks alleviates designers from some involved chores, but it also removes a potentially valuable source of design information. To help offset this loss of information, designers must understand the synthesis process very well, and they must be able to thoroughly analyze the post-synthesis circuit to make sure that all required specifications are met. This, in turn, requires rigorous use of simulators and other tools to recheck the design. These tools will be investigated in later labs.

Introduction to VHDL

In a schematic capture environment, a graphical symbol defines a given logic circuit by showing a "bounding box" as well as input and output connections. In VHDL, this same concept is used, only the bounding box must be explicitly typed into the text editor. The format for describing this bounding box requires an *entity* block with a corresponding *port* statement. The entity block (as shown in the example) gives the circuit a name and defines all input and

General Structure of a VHDL Source File

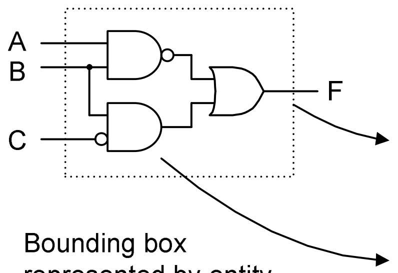
```
library ieee;
use ieee.std_logic_1164.all;

entity circuit_name is
  port (list of inputs, outputs and type);
end circuit_name;

architecture arch_name of circuit_name is
begin
    (statements defining circuit go here);
end arch_name;
```

output ports. Thus, the entity block in VHDL plays the same role as a symbol in a schematic capture environment. A VHDL circuit description also requires an *architecture* statement. The architecture statement defines circuit performance in the same manner as the "behind the scenes" simulation models define circuit performance in schematic capture programs. When VHDL code is simulated, these architecture statements are executed instead of the library-based simulation subroutines used in the schematic capture environment.

The general format for a VHDL circuit description is shown in the figure above. Required keywords have been shown in boldface, and text strings that the user must supply are shown in italics. The example below shows a schematic and corresponding VHDL code. The first two lines of the VHDL code establish the location of needed library elements. The actual function of these two lines will be explained later – for now, you can simply be type them as shown. By referring to the format and the following example, you can prepare VHDL code to describe the circuits required in this lab project that accompanies this module. In particular, you will need to define your circuit's input and output signal names in a port statement, and provide a description of circuit behavior in the architecture statement.



Bounding box represented by entity statement; behavior by architecture statement

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

entity Example is
  port (A,B,C    : in STD_LOGIC;
          Y      : out STD_LOGIC);
end Example

architecture behavioral of Example is
begin
  Y <= (not (A and B) or (B and not C));
end behavioral;
```

The port statement in the example code above defines the inputs A, B, and C, and the output Y as being of type std_logic. In VHDL, the std_logic type models signals that use wires in physical circuits. VHDL allows other data types (such as integers, characters, Boolean, etc.), but these more abstract types do not directly correspond to voltages on signals on wires. Rather, they are included so that a designer can think more about data flow than about electronics in the early stages of a design. These more abstract data types must be resolved to a std_logic type before the HDL description can be implemented. It is always possible to create a design using nothing but the std_logic type. In some cases, using only std_logic types can make the early stages of the design more time consuming, but in return, no translation of other data types is needed. For the next several labs, we use the std_logic type exclusively for all inputs and outputs.

Signal assignments

The act of designing a digital circuit can be simply described as creating new output signals based on some combination of input signals. From a computer system, to a controller inside of an appliance, to a media playback device, digital circuits process inputs from some source and produce useful outputs. Thus, the signal assignment operator in VHDL is the most fundamental operator.

The signal assignment operator ("<=") is used to indicate how an output signal is to be driven. Simple examples include A <= '1', meaning that the signal A gets assigned the logic value '1' (assumed to be LHV). Or, A <= B means the signal A gets assigned the signal B. Whenever a signal gets assigned a new value, the VHDL simulator requires that some amount of time passes before the signal is allowed to take the new value. This is because VHDL models circuits that use wires to carry signals – because the voltage on a wire cannot change instantaneously, a signal assignment in VHDL cannot happen instantaneously. This property differentiates VHDL from a computer language like "C". When a VHDL program is being executed, it is simulating a circuit that operates in real time. But "real time" to a computer is simply a count value stored in a memory location. In VHDL, if the time counter is at "430ns" when the assignment A <= '1' occurs, then A will not change to a '1' until the time value is incremented from 430ns. This is fundamentally different than a C program, where information transfer is not forced to occur over time.

Because time is a factor in every VHDL signal transaction, VHLD code is inherently *concurrent*, meaning that at any given time, several signal assignments may be pending. Cause-and-effect relationships are not a function of where a statement occurs in the VHDL code, but rather how time is modeled. For example, if in a C program variable X is storing a '1' and variable Y is storing a '2' when the statements X = Y followed by Z = X are encountered, Z = Y would be implied, and Z would immediately store a '2'. Not so in VHDL: if the statement A<=B is followed by C<=A, then C would store a '1' until enough time had passed to allow A to assume B's value, and then more time to allow C to assume A's value.

Signal assignment operators can assign an output signal a new value based on a *function* that operates on input signals (the signal on the left of the "<=" operator is the output, and those to the right are inputs). The basic logic operations are included as standard functions in VHDL tools, so that signal assignments such as "A <= C or D;" can be written (*and*, *or*, *nand*, *nor*, *xor*, *xnor*, and *not* may all be used). Thus, it is trivial to write VHDL code for basic logic circuits. VHDL contains other more powerful signal assignment operators, and more "built-in" functions, but these will be explored in later exercises. Note that all signal assignment statements in VHDL must be terminated with a semicolon.

Before beginning the lab procedure, you should follow the VHDL portion of the ISE/WebPack tutorial on the class website. Although the tutorial presents only the most basic features needed to design logic circuits using VHDL, it is sufficient for the needs of this lab. Later exercises will explore more features of the VHDL language, as well as more features available in WebPack.

<u>Using the ISE/WebPack VHDL tools</u>

To implement VHDL designs in the WebPack environment, a text editor is required to create the VHDL source file, a simulator is needed to check the results, and a synthesizer is needed to translate the source file to a form that can be downloaded to a chip. Other tools, like a floor-planner and/or timing analyzer, might also be needed for more complex designs (these tools will be discussed later). Any text editor can be used to create a VHDL source file. Xilinx supplies an editor with WebPack, and this editor uses colors and auto-indents to make the source file more readable (the WebPack editor is highly recommended). The lab project that accompanies this module provides a brief tutorial on creating VHDL designs, and then all subsequent labs will present further features of the VHDL language and Xilinx tool set. For now, we will start with a few simple projects to create a basic logic circuits like "Y<= (not A and B) or C".