Module 5: Introduction to VHDL

Revision: August 31, 2009

DIGILENT
www.digilentinc.com
215 E Main Suite D | Pullman, WA 99163
(509) 334 6306 Voice and Fax

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.

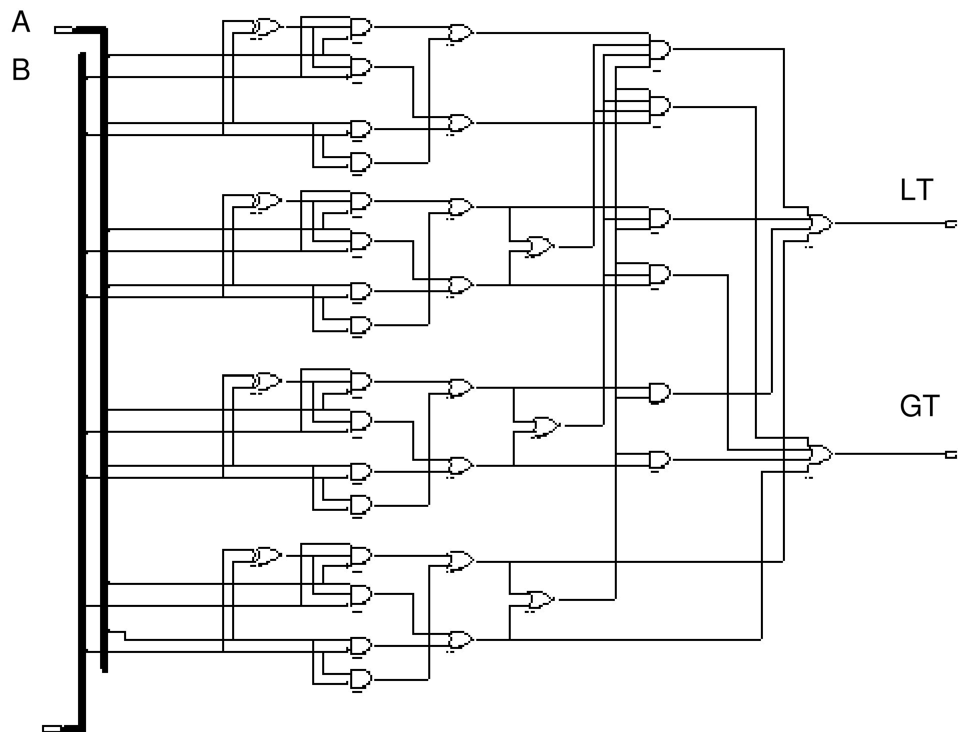
Digital design CAD tools can be placed in two major categories – the "front-end" tools that allow a design to be captured and simulated, and "back-end" tools that synthesize a design, map it to a particular technology, and analyze its performance (thus, front-end tools work mostly with virtual circuits, 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.

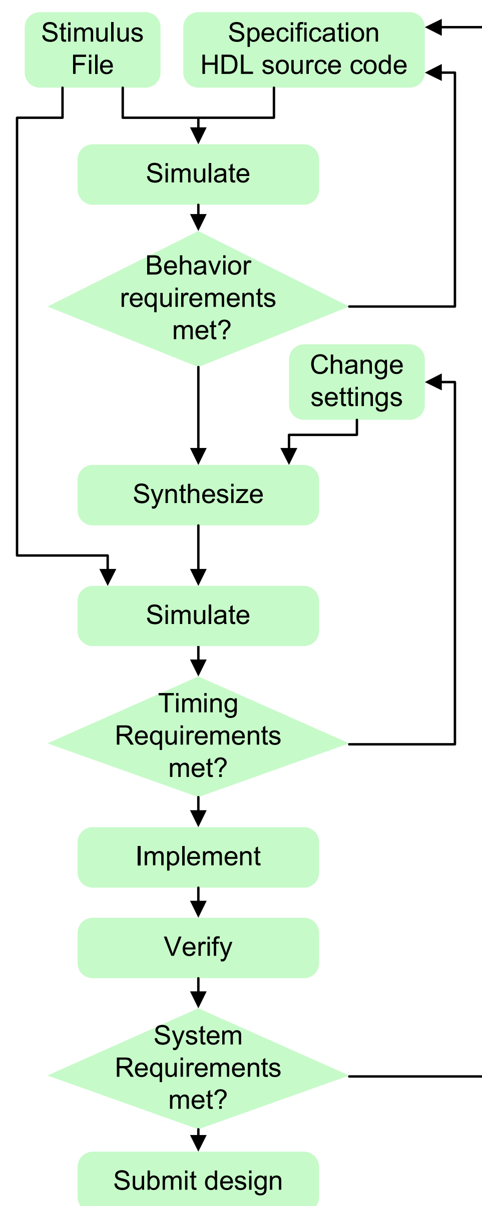


Schematic for a 4-bit magnitude comparator circuit

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 forwarded 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 subprograms. In a VHDL environment, no such simulation subprograms 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.

General Structure of a VHDL Source File

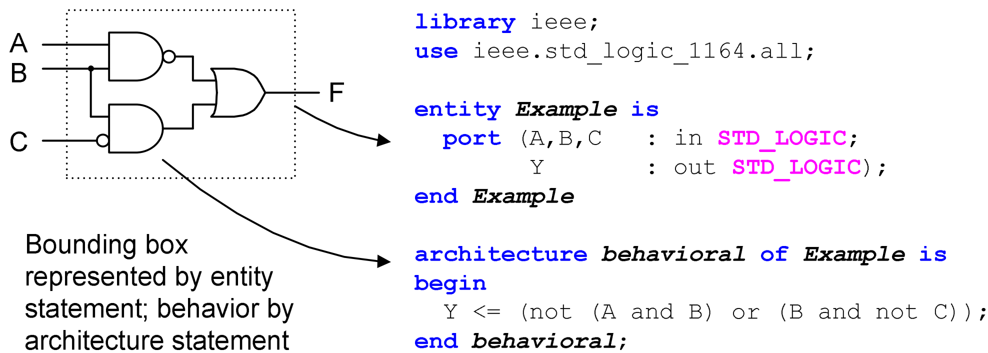
```
library ieee;
use ieee.std_logic_1164.all;

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;
```

Introduction to VHDL

DIGILENT®
www.digilentinc.com

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 bounding box is defined with an *entity* block and a corresponding *port* statement. The entity block (as shown in the example) gives the circuit a name and defines all input and output ports, and so plays the same role as a symbol in a schematic 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 environments. When VHDL code is simulated, these architecture statements are directly executed.

The general format for a VHDL circuit description is shown in the figure above. Required keywords have been shown in blue boldface, and key text strings 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 (the *library* and *use* statements) establish the location of needed library elements. The function of these statements will be explained later – for now, assume you will include library and use statements at the start of every VHDL source file.



The port statement in the example 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 signal wires. Rather, they allow designers to think about data and algorithms instead of circuits in the early stages of a design. The trade-off is that extra VHDL code must be written to resolve these more abstract data types to the std_logic type prior to synthesis. Since it is always possible to create a design using nothing but the std_logic type, we will put off a discussion of the more abstract types until a later module.

By referring to the example above, you can immediately prepare VHDL code to describe basic logic circuits. But to use VHDL effectively and to create more complex circuits, you must understand more about the language, starting with the most basic VHDL operation – signal assignments.

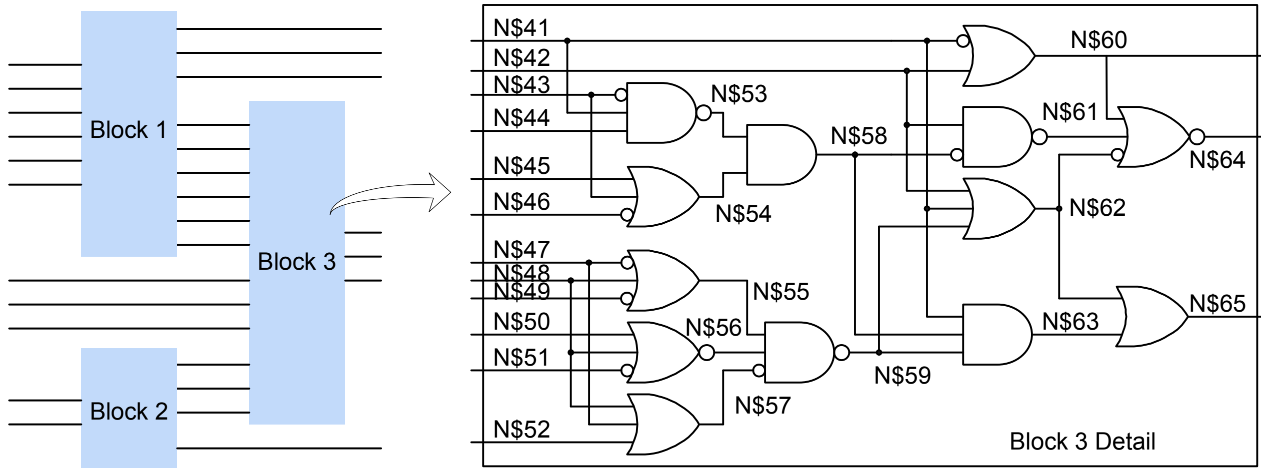
Signal assignments

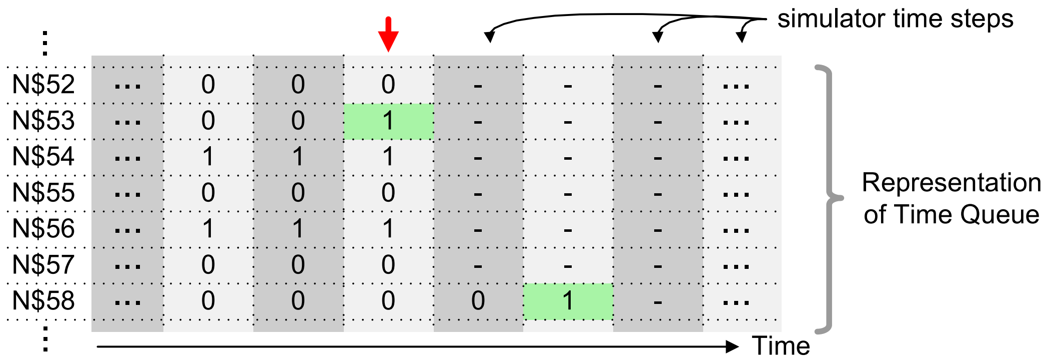
A digital circuit can be simply described as an electronic device that drives output signals to a '0' or '1' based on some function or combination of input signals. Whether used in a computer system, an appliance controller, a media playback device, or any other device, a digital circuit processes input signals from some source and produces output signals that are useful in some context. Restated, output signals from digital circuits are assigned new values based on functions or combinations of input signals. Thus, the signal assignment operator ("<=") in VHDL is the most fundamental operator.

A synthesizer uses VHDL source code as the basis for a physical circuit, and signal assignment statements provide the information needed to build the circuit. A simulator uses VHDL source code to model circuits, and signal assignments describe how to drive signals using a model of physical time. Because signal assignments model signals as they change over physical time, they are different than variable assignments in a language like C.
Simple examples of signal assignments include the statements “A <= B;” meaning that the signal A gets assigned the value carried on signal B, and “A <= B or C;” meaning the signal A gets assigned the result of the logical OR of signal B and C (signals to the left of the assignment operator are outputs, and signals to the right are inputs). When a signal gets assigned a new value, the simulator requires that some amount of time pass before the signal is allowed to take the new value. This is because VHDL describes circuits that use transistors and wires to transport signal voltages, and no voltage, not even the voltage on an infinitesimally small wire, can change instantaneously.

The simulator models physical time so it can keep track of exactly when signals change state, and ensure that any resulting output signal changes are made at the appropriate time. As shown in the figure below, the simulator assigns a unique identifier to every circuit node, and builds a data structure to hold the state of every node for the duration of the simulation. The simulator models time as a sequence of very small steps (generally 10ps, or 10 trillionths of a second), and signal states are held for every circuit node for every time step. The simulation progresses as the simulator increments the “current time” through all the time steps in a given simulation. At any given time, all circuit nodes have a defined value stored in the data structure. Some circuit nodes may have just changed state at the current time, and this is a call to action for the simulator: whenever a signal changes state, any “downstream” circuits that include that signal as an input must be evaluated to see if a new output value results. If a new output state results from a signal change, it cannot take affect immediately. Rather, it must be scheduled to take effect at a later time.

In the figure below, current time is indicated by a red arrow; entries to the left are in the past and so have values, and entries to the right are in the future and so are undefined. At the current time, N$53 has just changed to a ‘1’. Because of that change, N$58 must change to a ‘1’, but it cannot change at the current time. Rather, a change on N$58 is scheduled for a later time step, and a ‘1’ is entered into the time queue at a future time.





N$52	...	0	0	0	-	-	-	...
N$53	...	0	0	1	-	-	-	...
N$54	...	1	1	1	-	-	-	...
N$55	...	0	0	0	-	-	-	...
N$56	...	1	1	1	-	-	-	...
N$57	...	0	0	0	-	-	-	...
N$58	...	0	0	0	0	1	-	...

As the simulation proceeds, the current time is incremented (and so the red arrow moves to the right). Eventually, the current time will reach the time of the scheduled change to N$58. When the simulator advances to that time, any circuit nodes that might change state because of the change on N$58 will be evaluated (for example, N$61 and N$63), and if necessary, output signal changes will be scheduled on those nodes for some future time. The difference in time between the current time and a scheduled change represents the time taken to move the voltage signal through the transistors and wires of the logic circuit.

At any given time step, the simulator evaluates all nodes that might possibly change state due to all signal changes recorded in the Time Queue. The simulator current time will only advance after all current signal changes have been evaluated and any resulting signals changes scheduled.

This property of requiring some amount of time to pass before a signal assignment can take affect differentiates VHDL from a computer language like “C”. When a VHDL program is being “executed”, it is modeling a circuit that operates in real, physical time, and the “current” physical time simply a value stored in a variable. At any given time step, all signal assignment operations in a source file are analyzed to see if an input to the assignment operator has changed. Any assignment operations with current input signal changes are executed, and any resulting signal changes are scheduled for a later time. This makes VHDL concurrent, because it does not matter in what order assignment statements appear in a VHDL source file; rather, assignment operations are performed whenever their input signals change state. This is fundamentally different than a C program, where the order of statements does matter, where information transfer is not forced to occur over time, and where variable assignments take effect immediately.

Because time is modeled in every VHDL assignment, cause-and-effect relationships are not a function of where a statement occurs in VHDL code, and assignment statements can occur in any

order without effecting simulation (or synthesis) results. For example, the two VHDL code segments shown below would produce identical circuits during synthesis and would simulate identically the same; that is, the B signal would take a new value at the same time and for the same reasons (i.e., X or Y going to a ‘1’). This is different than the C code, where B would take on the value X or Y immediately in the code on the left, and B would not take on the value X or Y in the code on the right.

```
        VHDL code                          C code
A <= X or Y;   B <= A;            A = X | Y;    B = A;
B <= A;        A <= X or Y;       B = A;        A = X | Y;
```

Functions and signal assignments

Functions play the same role in VHDL as in other computer language environments. They encapsulate commonly used code into a reusable module, and they are written to operate on parameters rather than signal or variable names. They can be used in any situation by “calling” them along with parameters specific to the contextual need. Standard VHDL tool installations provide many functions in standard IEEE-specified libraries, and these functions can be use directly (provided the library and use statements are included in your source file as mentioned earlier). It is also possible to write new VHDL functions for use in particular projects. Creating new functions typically is typically not beneficial until projects become quite complex, as so new function creation will not be addressed until later.

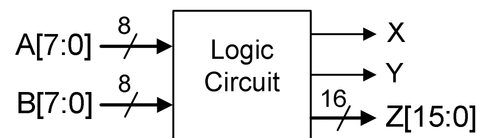
Basic logic functions in VHDL are not a part of the language itself, but they are supplied as functions stored in the IEEE “standard logic 1164” library distributed with virtually every VHDL system. Common logic functions such as and, or, nand, nor, xor, xnor, and not are included along with several others, and they can be used in assignment statements to assign a new output value based on a function of input signals. As examples, signal assignments like those shown below can be written.

```
A <= C or D;          Y <= not (A and B) or (C nand D);
```

As shown in the example statements, parenthesis should always be used to make function precedence clear. If no parentheses are used, a predefined function precedence is used, but it can make source code difficult to follow (basic precedence is defined as: not, and, or, and then xor/xnor).

Using library-supplied logic functions makes it very easy 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 modules. Note that all signal assignment statements in VHDL must be terminated with a semicolon.

Binary Numbers as Signals (busses)



In a digital system, all information must be encoded as signals that carry either a logic high voltage (a ‘1’) or logic low voltage (a ‘0’). Individual signals are often grouped together to form logical units called “buses” for the purpose of representing a binary number. For example, a group of four signals can be viewed as a logical grouping that can represent the first ten four-bit binary numbers (0000 – 1001), or all 16 hexadecimal digits (0000 – 1111). In the block diagram above, the two inputs A and B and the output Z are shown as bold lines with hash marks cutting them, and the

number “8” near the hash mark. This notation is used to show groups of logic signals that are to be considered as a bus carrying a binary number. In this case, A and B are 8-bit busses, meaning they each are composed of 8 wires that, when taken together, represent a single 8-bit binary number. The output bus Z is a 16-bit bus that can transport a 16-bit number, and the outputs X and Y are simply 1-bit signal wires. Busses are typically used when circuit components produce or consume binary numbers.

Nearly all schematic capture tools represent busses in the same way. The bus is given a name, followed by two numbers enclosed in parenthesis and separated by a colon. For example, a bus labeled “A(7:0)” would apply to an 8-bit bus named “A”. Each wire in the bus has a unique name that is formed by concatenating the bus “root” name with a number in the indicated range. For instance, A0 would refer to the 0th wire in the bus, and A4 would refer to the 4th wire in the bus. Note that although the use of buses makes it easier to refer to collections of similar signals, it is always possible to represent those signals as individual wires (and note also that if you were to construct the circuit on a circuit board, you’d need 8 individual wires for each bus). In most schematic capture tools, busses can be added like wires. A bus-wire is first drawn using the “add bus” tool, and the bus is given a name using the convention “bus_name (MSB:LSB)”, where MSB means “most significant bit”, or the highest-numbered bit, and LSB means least-significant bit. Then individual wires can added and connected to the bus, and named as discussed above so that it is clear which bus line is connected to a given signal.

In VHDL, busses can be defined using the “std_logic_vector” type. For example, an 8-bit input bus named A can be defined in a VHDL port statement by typing “A: in std_logic_vector (7 downto 0)”. The entire bus can be referred to as simply “A”, and signal number 5 can be referred to as A(5). Many VHDL assignments can refer to busses as well as signals. For example, the statement “Y <= A or B;” will drive the Y signal with the logical combination of A and B if Y, A, and B are identified as std_logic signals; or it will drive every bit of the Y bus with the bit-wise OR’ing of the A and B bus signals if Y,A, and B are defined as std_logic_vectors. As an example, if Y, A and B are all buses, and A = “11110000” and B = “10101010”, then the statement “Y<=A and B;” will drive Y to “10100000”.

VHDL Test Benches

Another key advantage to creating circuits using VHDL is that you can create a separate VHDL source file called a “test bench” to provide stimulus input for the design that can be directly executed by the simulator. In practice, creating a VHDL test bench stimulus file is by far the simplest and most efficient way to simulate any given source file. When compared with a wave-form editor or other means of creating stimulus input, using a test bench makes it easier to create a wider range of stimulus inputs, particularly with respect to the timing of input signals.

A test bench source file can be created and added to a project in the same manner as any other source file, except that it must be identified as a test bench file instead of a VHDL module. An example of a test bench suitable for use in the exercises is presented on the following page. By following this example, you can readily create test bench files for any logic circuit.

A test bench is an entity-architecture pair that looks similar to any other VHDL source file, except that the entity statement is empty (see example below), and the VHDL source file that is to be simulated must be “instantiated” as a component. In later modules, more information will be presented about test benches, their use, and their capabilities.

A VHDL test bench is our first look at a more general VHDL coding technique called "structural design" that is commonly used for more complex designs. Any VHDL source file can be used as a component in any other source file, much like any schematic design can be wrapped in a macro symbol and used in another design. When "lower level" VHDL source files are used as components in "higher-level" source files, a more hierarchical and structural design results, and often this is the best way to attack complex designs. A VHDL test bench uses structural VHDL as a convenient way to generate a clear and concise mapping of stimulus input to source file.

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

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.

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".

```
Library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
```

Standard file header containing library and package definitions

```
entity lab5test_bench is
end lab5test_bench;
```

An “empty” entity statement required for all test bench source files. The entity name can be any legal string.

```
architecture test of lab5test_bench is

  component ex1
      port( a, b, c  : in std_logic;
            y  : out std_logic);
  end component;
```

The entity under test (EUT) must be declared as a component. The port must exactly match the port statement from the EUT.

```
  signal a, b, c, y : std_logic;
```

All signals that attach to entity port pins must be declared as signals.

```
begin

  EUT: ex1 port map(a => a,
                    b => b,
                    c => c,
                    y => y);
```

The EUT must be instantiated. The port map statement maps the declared signals to the port pins of the EUT. It is common to use matching signal and port pin names.

```
  process begin

    a <= '0';
    b <= '0';
    c <= '0';
    wait for 100 ns;
    a <= '1';
    wait for 100 ns;
    b <= '1';
    wait for 100 ns;
    c <= '1';
    wait for 100 ns;
    a <= '0';
    wait for 100 ns;
    b <= '0';
    wait for 100 ns;
    c <= '0';

  end process;
end test;
```

Statements to define input stimulus are placed in a process statement so that the “wait” statement can be used to control the passage of time.