

Hardware Description Languages

4

4.1 INTRODUCTION

Thus far, we have focused on designing combinational and sequential digital circuits at the schematic level. The process of finding an efficient set of logic gates to perform a given function is labor intensive and error prone, requiring manual simplification of truth tables or Boolean equations and manual translation of finite state machines (FSMs) into gates. In the 1990's, designers discovered that they were far more productive if they worked at a higher level of abstraction, specifying just the logical function and allowing a *computer-aided design* (CAD) tool to produce the optimized gates. The specifications are generally given in a *hardware description language* (HDL). The two leading hardware description languages are *Verilog* and *VHDL*.

Verilog and VHDL are built on similar principles but have different syntax. Discussion of these languages in this chapter is divided into two columns for literal side-by-side comparison, with Verilog on the left and VHDL on the right. When you read the chapter for the first time, focus on one language or the other. Once you know one, you'll quickly master the other if you need it.

Subsequent chapters show hardware in both schematic and HDL form. If you choose to skip this chapter and not learn one of the HDLs, you will still be able to master the principles of computer organization from the schematics. However, the vast majority of commercial systems are now built using HDLs rather than schematics. If you expect to do digital design at any point in your professional life, we urge you to learn one of the HDLs.

4.1.1 Modules

A block of hardware with inputs and outputs is called a *module*. An AND gate, a multiplexer, and a priority circuit are all examples of hardware modules. The two general styles for describing module functionality are

- 4.1 [Introduction](#)
- 4.2 [Combinational Logic](#)
- 4.3 [Structural Modeling](#)
- 4.4 [Sequential Logic](#)
- 4.5 [More Combinational Logic](#)
- 4.6 [Finite State Machines](#)
- 4.7 [Parameterized Modules*](#)
- 4.8 [Testbenches](#)
- 4.9 [Summary](#)
- [Exercises](#)
- [Interview Questions](#)

behavioral and *structural*. Behavioral models describe what a module does. Structural models describe how a module is built from simpler pieces; it is an application of hierarchy. The Verilog and VHDL code in HDL Example 4.1 illustrate behavioral descriptions of a module that computes the Boolean function from Example 2.6, $y = \bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c} + a\bar{b}c$. In both languages, the module is named `sillyfunction` and has three inputs, `a`, `b`, and `c`, and one output, `y`.

HDL Example 4.1 COMBINATIONAL LOGIC

Verilog

```
module sillyfunction(input a, b, c,
                    output y);

    assign y = ~a & ~b & ~c |
              a & ~b & ~c |
              a & ~b & c;

endmodule
```

A Verilog module begins with the module name and a listing of the inputs and outputs. The `assign` statement describes combinational logic. `~` indicates NOT, `&` indicates AND, and `|` indicates OR.

Verilog signals such as the inputs and outputs are Boolean variables (0 or 1). They may also have floating and undefined values, as discussed in Section 4.2.8.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sillyfunction is
    port(a, b, c: in STD_LOGIC;
         y: out STD_LOGIC);
end;

architecture synth of sillyfunction is
begin
    y <= ((not a) and (not b) and (not c)) or
         (a and (not b) and (not c)) or
         (a and (not b) and c);
end;
```

VHDL code has three parts: the `library use` clause, the `entity` declaration, and the `architecture body`. The `library use` clause is required and will be discussed in Section 4.2.11. The `entity` declaration lists the module name and its inputs and outputs. The `architecture body` defines what the module does.

VHDL signals, such as inputs and outputs, must have a *type declaration*. Digital signals should be declared to be `STD_LOGIC` type. `STD_LOGIC` signals can have a value of '0' or '1', as well as floating and undefined values that will be described in Section 4.2.8. The `STD_LOGIC` type is defined in the `IEEE.STD_LOGIC_1164` library, which is why the library must be used.

VHDL lacks a good default order of operations, so Boolean equations should be parenthesized.

A module, as you might expect, is a good application of modularity. It has a well defined interface, consisting of its inputs and outputs, and it performs a specific function. The particular way in which it is coded is unimportant to others that might use the module, as long as it performs its function.

4.1.2 Language Origins

Universities are almost evenly split on which of these languages is taught in a first course, and industry is similarly split on which language is preferred. Compared to Verilog, VHDL is more verbose and cumbersome,

Verilog

Verilog was developed by Gateway Design Automation as a proprietary language for logic simulation in 1984. Gateway was acquired by Cadence in 1989 and Verilog was made an open standard in 1990 under the control of Open Verilog International. The language became an IEEE standard¹ in 1995 (IEEE STD 1364) and was updated in 2001.

VHDL

VHDL is an acronym for the *VHSIC Hardware Description Language*. VHSIC is in turn an acronym for the *Very High Speed Integrated Circuits* program of the US Department of Defense.

VHDL was originally developed in 1981 by the Department of Defense to describe the structure and function of hardware. Its roots draw from the Ada programming language. The IEEE standardized it in 1987 (IEEE STD 1076) and has updated the standard several times since. The language was first envisioned for documentation but was quickly adopted for simulation and synthesis.

as you might expect of a language developed by committee. U.S. military contractors, the European Space Agency, and telecommunications companies use VHDL extensively.

Both languages are fully capable of describing any hardware system, and both have their quirks. The best language to use is the one that is already being used at your site or the one that your customers demand. Most CAD tools today allow the two languages to be mixed, so that different modules can be described in different languages.

4.1.3 Simulation and Synthesis

The two major purposes of HDLs are logic *simulation* and *synthesis*. During simulation, inputs are applied to a module, and the outputs are checked to verify that the module operates correctly. During synthesis, the textual description of a module is transformed into logic gates.

Simulation

Humans routinely make mistakes. Such errors in hardware designs are called *bugs*. Eliminating the bugs from a digital system is obviously important, especially when customers are paying money and lives depend on the correct operation. Testing a system in the laboratory is time-consuming. Discovering the cause of errors in the lab can be extremely difficult, because only signals routed to the chip pins can be observed. There is no way to directly observe what is happening inside a chip. Correcting errors after the system is built can be devastatingly expensive. For example, correcting a mistake in a cutting-edge integrated circuit costs more than a million dollars and takes several months. Intel's infamous FDIV (floating point division) bug in the Pentium processor forced the company to recall chips after they had shipped, at a total cost of \$475 million. Logic simulation is essential to test a system before it is built.

¹ The Institute of Electrical and Electronics Engineers (IEEE) is a professional society responsible for many computing standards including WiFi (802.11), Ethernet (802.3), and floating-point numbers (754) (see Chapter 5).

The term “bug” predates the invention of the computer. Thomas Edison called the “little faults and difficulties” with his inventions “bugs” in 1878. The first real computer bug was a moth, which got caught between the relays of the Harvard Mark II electro-mechanical computer in 1947. It was found by Grace Hopper, who logged the incident, along with the moth itself and the comment “first actual case of bug being found.”



Source: Notebook entry courtesy Naval Historical Center, US Navy; photo No. NII 96566-KN

Figure 4.1 Simulation waveforms

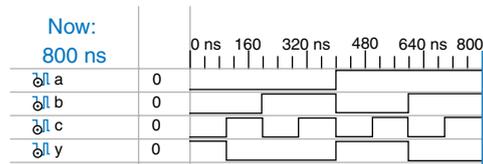


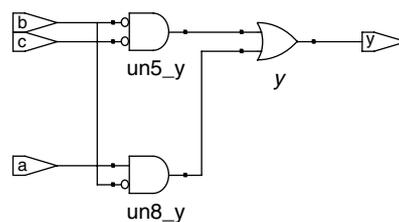
Figure 4.1 shows waveforms from a simulation² of the previous `sillyfunction` module demonstrating that the module works correctly. `y` is TRUE when `a`, `b`, and `c` are 000, 100, or 101, as specified by the Boolean equation.

Synthesis

Logic synthesis transforms HDL code into a *netlist* describing the hardware (e.g., the logic gates and the wires connecting them). The logic synthesizer might perform optimizations to reduce the amount of hardware required. The netlist may be a text file, or it may be drawn as a schematic to help visualize the circuit. Figure 4.2 shows the results of synthesizing the `sillyfunction` module.³ Notice how the three three-input AND gates are simplified into two two-input AND gates, as we discovered in Example 2.6 using Boolean algebra.

Circuit descriptions in HDL resemble code in a programming language. However, you must remember that the code is intended to represent hardware. Verilog and VHDL are rich languages with many commands. Not all of these commands can be synthesized into hardware. For example, a command to print results on the screen during simulation does not translate into hardware. Because our primary interest is

Figure 4.2 Synthesized circuit



² The simulation was performed with the Xilinx ISE Simulator, which is part of the Xilinx ISE 8.2 software. The simulator was selected because it is used commercially, yet is freely available to universities.

³ Synthesis was performed with Synplify Pro from Synplicity. The tool was selected because it is the leading commercial tool for synthesizing HDL to field-programmable gate arrays (see Section 5.6.2) and because it is available inexpensively for universities.