HARDWARE DESCRIPTION LANGUAGES, A COMPARATIVE APPROACH

Iuliana PATENTARU, Alin Dan POTORAC
iuliap@eed.usv.ro, alinp@eed.usv.ro
"Stefan cel Mare" University of Suceava
str.Universitatii nr.9, RO-5800 Suceava

Abstract. *The Hardware Description Languages (HDL) are the only efficient solution for complex logical design. In the last years some specific HDLs were imposed by the logical design market, most used being Verilog and VHDL. This material contains two parts. The first part takes a view of VHDL and Verilog by comparing their similarities and contrasting their differences. The second part contains an example of a model of a 4-bit multiplexer in VHDL and Verilog. Some advantages of each one are revealed based on a similar logical structure simulation.*
Keywords: *Hardware Description Language, Verilog, VHDL, logical design, modeling, simulation*

1. Introduction

As literature is mentioning [3], [4], [6], [7] there are now two industry standard hardware description languages, VHDL and Verilog. The complexity of ASIC and FPGA designs has meant an increase in the number of specialist design consultants with specific tools and with their own libraries of macro and mega cells written in either VHDL or Verilog. As a result, it is important that designers know both VHDL and Verilog specificity as available EDA vendors tools are covering both approaches.

Verilog was introduced in 1985 by Gateway Design System Corporation, now a part of Cadence Design Systems, Inc.'s Systems Division. Until May, 1990, with the formation of Open Verilog International (OVI), Verilog HDL was a proprietary language of Cadence.
Cadence bought Gateway in 1989 and opened Verilog to the public domain in 1990. It became IEEE standard 1364 in December 1995.

VHDL (Very high speed integrated circuit Hardware Description Language) arose out of the United States Government's Very High Speed Integrated Circuits (VHSIC) program, initiated in 1980. In the course of this program, it became clear that there was a need for a standard language for describing the structure and function of integrated circuits (ICs). Hence the VHSIC Hardware Description Language (VHDL) was developed, and subsequently adopted as a standard by the Institute of Electrical and Electronic Engineers (IEEE) in the US.

2. Verilog versus VHDL, an overview

Verilog HDL [1], [6] allows a hardware designer to describe designs at a high level of abstraction such as at the architectural or behavioural level as well as the lower implementation levels (i. e. , gate and switch levels) leading to Very Large Scale Integration (VLSI) Integrated Circuits (IC) layouts and chip fabrication. A primary use of HDLs is the simulation of designs before the designer must commit to fabrication.

VHDL [2], [3], [7] is a language for describing digital electronic systems. It is designed to fill a number of needs in the design process. Firstly, it allows description of the structure of a design that is how it is decomposed into sub-designs, and how those sub-designs are interconnected. Secondly, it allows the specification of the function of designs using familiar programming language forms. Thirdly, as a result, it allows a design to be simulated before being manufactured, so that designers can quickly compare alternatives and test for correctness

without the delay and expense of hardware prototyping.
Hardware structure can be modeled equally effectively in both VHDL and Verilog. When modeling abstract hardware, the capability of VHDL can sometimes only be achieved in Verilog when using the PLI.

As some articles [4] are pointing, there are some structural and conceptual differences on Verilog and VHDL but also a lot of similarities, as shown below.

Regarding from the point of view of the ***lexical conventions***, Verilog is close to the programming language C++. Comments are designated by // to the end of a line or by /* to */ across several lines. Keywords, e. g., *module,* are reserved and in all lower case letters. The language is case sensitive, meaning upper and lower case letters are different. Spaces are important in that they delimit tokens in the language.
Meanwhile, in *VHDL* Comments start with two adjacent hyphens ('--') and extend to the end of the line. They have no part in the meaning of a VHDL description. The case of letters is not considered significant, so the identifiers *mux* and *Mux* are the same. Underline characters in identifiers are significant, so *This_Name* and *ThisName* are different identifiers.

As the ***program structure*** level the Verilog language describes a digital system as a set of modules. Each of these modules has an interface to other modules to describe how they are interconnected. Usually we place one module per file but that is not a requirement. The modules may run concurrently, but usually we have one top level module which specifies a closed system containing both test data and hardware models. The top level module invokes instances of other modules. Modules can represent pieces of hardware ranging from simple gates to complete systems, e. g., a microprocessor.
A VHDL digital electronic system can be described as a module with inputs and/or outputs. One way of describing the function of a module is to describe how it is composed of sub-modules. Each of the sub-modules is an *instance* of some entity, and the ports of the instances are connected using *signals*. More complex behaviours cannot be described purely as a function of inputs. In systems with feedback, the outputs are also a function of time. VHDL solves this problem by allowing description of behaviour in the form of an executable program.

Since the purpose of Verilog HDL is to model digital hardware, the primary ***data types*** are for modeling registers (*reg*) and wires (*wire*). The *reg* variables store the last value that was procedurally assigned to them whereas the *wire* variables represent physical connections between structural entities such as gates. A *wire* does not store a value. A *wire* variable is really only a label on a wire. For the convenience of the designer, Verilog has several data types which do not have a corresponding hardware realization. These data types include *integer*, *real* and *time*. The data types *integer* and *real* behave pretty much as in other languages, e. g., C.
VHDL data types provide a number of basic, or *scalar*, types, and a means of forming *composite* types. The scalar types include numbers, physical quantities, and enumerations (including enumerations of characters), and there are a number of standard predefined basic types. The composite types provided are arrays and records.

Some minor differences could be finding at the available ***operators***' level. The majority of operators are the same between the two languages. Verilog does have very useful unary reduction operators that are not in VHDL. A loop statement can be used in VHDL to perform the same operation as a Verilog unary reduction operator. VHDL has the mod operator that is not found in Verilog. Same on ***procedures and tasks***: *Verilog* does not allow concurrent task calls, *VHDL* allows concurrent procedure calls.

Like ***compilation*** the Verilog language is still rooted in its native interpretative mode.

Compilation is a means of speeding up simulation, but has not changed the original nature of the language. As a result care must be taken with both the compilation order of code written in a single file and the compilation order of multiple files. Simulation results can change by simply changing the order of compilation. On *VHDL,* multiple *design-units* (*entity/architecture* pairs), that reside in the same system file, may be separately compiled. However, it is good design practice to keep each design unit in its own system file in which case separate compilation should not be an issue.

On Verilog, functions and procedures used within a model must be defined in the *module*. To make functions and procedures generally accessible from different *module* statements the functions and procedures must be placed in a separate system file and included using *include* compiler directive. *VHDL* procedures and functions may be placed in a *package* so that they are available to any *design-unit* that wishes to use them.

Concerning the ***libraries***, there is no concept of a library in Verilog. This is due to it's origins as an interpretive language while in *VHDL*, a library is a store for compiled entities, architectures, packages and configurations. Libraries are useful for managing multiple design projects.

The Verilog language was developed with gate level modeling and has very good constructs for modeling at this level and for modeling the cell primitives of ASIC and FPGA libraries. Examples include User Defined Primitives (UDP), truth tables and the specify block for specifying timing delays across a module. In *VHDL* simple two input logical operators are built and they are: NOT, AND, OR, NAND, NOR, XOR and XNOR. Any timing must be separately specified using the *after* clause.

There are also more constructs and features for high-level modelings in VHDL than there are in Verilog. Abstract data types can be used along with the following statements: package statements for model reuse, configuration statements for configuring design structure, generate statements for replicating structure, generic statements for generic models that can be individually characterized. Except for being able to parameterize models by overloading parameter constants, there is no equivalent to the high-level VHDL modeling statements in Verilog

Using one language or another is more a matter of coding style and experience than language feature. *Verilog* is more like C because its constructs are based approximately 50% on C and 50% on Ada. *VHDL* is a concise and verbose language; its roots are based on Ada. For this reason an C programmer may prefer Verilog over VHDL. Although a programmer of both C and Ada may find the mix of constructs somewhat confusing at first.

Starting with zero knowledge of either language, *Verilog* is probably the easiest to learn and understand. This assumes the Verilog compiler directive language for simulation and the PLI language is not included. If these languages are included they can be looked upon as two additional languages that need to be learned.
VHDL may seem less intuitive at first for two reasons. First, it is very strongly typed; a feature that makes it robust and powerful for the advanced user after a longer learning phase. Second, there are many ways to model the same circuit, especially those with large hierarchical structures.

3. Modeled example in VHDL and Verilog

In this section we will look at a small example of a Verilog and VHDL descriptions of a 4-bit multiplexer to give you a feel for the language and how it is used.

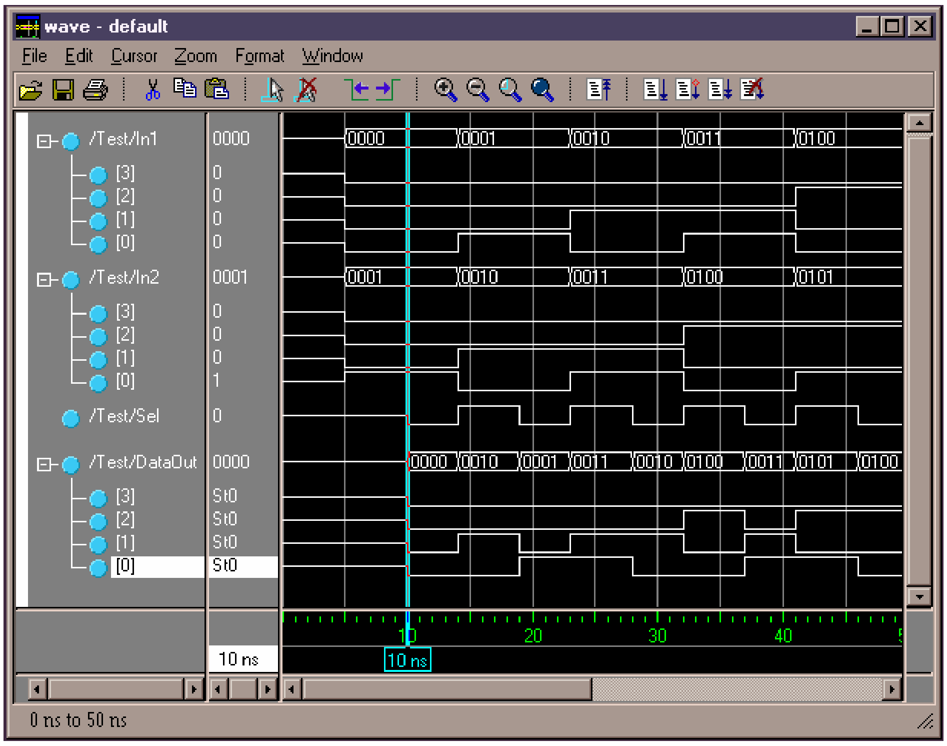

Figure 1. Verilog Model Simulation

We start the description of a circuit by specifying its external interface, which includes a description of its ports. The multiplexer might be defined so that: when the signal *Sel* is ”*0*” logic multiplexer’s output (*y*) is equal with first input (*a*), else with second input (*b*).

Verilog Model

In *module MUX*, we declared *In1* and *In2* as 4-bit inputs, *DataOut* a 4-bit output and register and *Sel* is a 1-bit input. Inside of the module we have one “*always*” construct. The *always* construct is the same as the *initial* construct except that it loops forever as long as the simulation runs. Within the *initial* construct, statements are executed sequentially much like in C or other traditional imperative programming languages.
The notation *#5* means to execute the statement after delay of 5 units of simulated time, before calling the system task *$stop* and stop the simulation. Notice that all the statements in the second *initial* are done at time = 0, since there are no delay statements, i. e., *#<integer>*.

The semantics of the *module* construct in Verilog is very different from subroutines, procedures and functions in other languages. *A module is never called!* A module is instantiated at the start of the program and stays around for the life of the program. A Verilog module instantiation is used to model a hardware circuit. Each time a module is instantiated, we give its instantiation a name.
Verilog model for described MUX is shown below.

```
module MUX (In1, In2, Sel, DataOut);

input[3:0] In1;
input[3:0] In2;
input Sel;
output[3:0] DataOut;
reg[3:0] DataOut;

//--------------------------------------------
// Model Multiplexer algorithm
//--------------------------------------------

always@(In1 or In2 or Sel)
begin
 if(Sel==0)
```

```
      DataOut=In1;
  else
  if(Sel==1)
        DataOut =In2;
end

endmodule

//---------------------------------------------
// Test Multiplexer algorithm
//---------------------------------------------

module Test ();

reg[3:0] In1;
reg[3:0] In2;
reg Sel;
wire[3:0] DataOut;
integer i;

MUX Multiplexer(In1,In2,Sel,DataOut);

initial
begin
i=8;
#5 In1=0;
   In2=1;
for(i=0;i<=8;i=i+1)
begin
        #5 Sel=0;
        #4 Sel=1;
          In1=In1+1;
          In2=In2+1;
end
end

initial
#1000 $stop;

endmodule
```

VHDL Model

A digital system is usually designed as a hierarchical collection of modules. Each module has a set of ports which constitute its interface to the outside world. In VHDL, an *entity* is such a module which may be used as a component in a design, or which may be the top level module of the design.

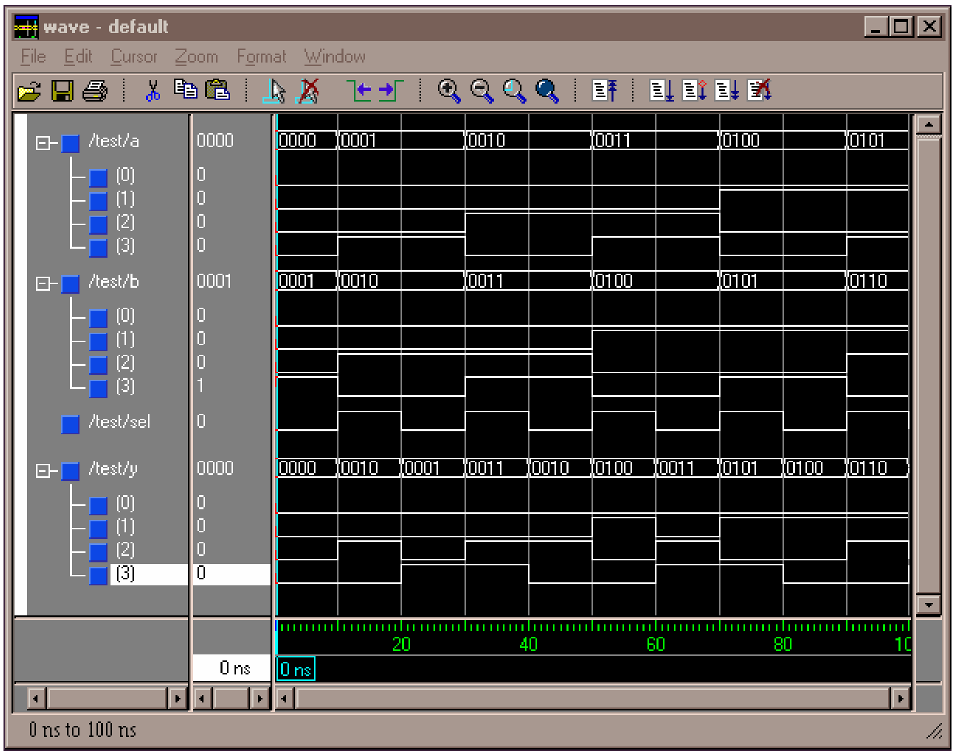

Figure 2. VHDL Model Simulation

The entity may include specification of *generic constants*, which can be used to control the structure and behaviour of the entity, and *ports*, which channel information into and out of the

entity. An implementation of the entity is described in an architecture body. There may be more than one architecture body corresponding to a single entity specification, each of which describes a different view of the entity. The architecture contains an instance of the named component, with actual values specified for generic constants, and with the component ports connected to actual signals or entity ports.
VHDL model for described MUX is shown below.

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

entity mux is
        port(a,b : in std_logic_vector(0 to 3);
                sel : in std_logic;
y : out std_logic_vector(0 to 3));
end mux;

architecture amux of mux is
begin
l:process(a,b,sel)
begin
if(sel='0') then
        y<=a;
else
        y<=b;
end if;
end process;
end amux;
-- ------------------------------------------------
-- Model Multiplexer algorithm
-- ------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity test is
end test;

architecture test of test is
   signal a,b:std_logic_vector(0 to 3);
   signal sel:std_logic:='0';
   signal y:std_logic_vector(0 to 3);

component muxc
port(a,b : in std_logic_vector(0 to 3);
      sel : in std_logic;
      y : out std_logic_vector(0 to 3));
end component;

for all:muxc use entity work.mux(amux);

begin
dut:muxc port map(a,b,sel,y);

sel<=not sel after 10 ns;

a<="0000", "0001" after 10 ns, "0010" after 30 ns,
"0011" after 50 ns,"0100" after 70 ns, "0101" after
90 ns,"0110" after 110 ns, "0111" after 130
ns,"1000" after 150 ns;

b<="0001", "0010" after 10 ns, "0011" after 30
ns,"0100" after 50 ns, "0101" after 70 ns, "0110"
after 90 ns, "0111" after 110 ns,"1000" after 130 ns,
"1001" after 150 ns;

end test;
```

4. Conclusions

The reasons for the importance of being able to model hardware in both VHDL and Verilog have been discussed. VHDL and Verilog has been extensively compared and contrasted in a neutral manner and a tutorial has been posed as a problem and solution to demonstrate the issues above.

References

[1] Hyde, D.C. (1997) Handbook on Verilog HDL, Bucknell University, Lewisburg, USA
[2] Nicula, D. (2000) Proiectarea Sistemelor Digitale Implementate cu Dispozitive Programabile, Bucuresti, Ed.Tehnica
[3] Pellerin D. (1998) An Introduction to HDLs for Simulation and Synthesis, Protel Technology Inc., Provo, USA
[4] Smith, D.J. (1996) *VHDL & Verilog Compared and Contrasted*, 33rd Design Automation Conference, VeriBest Incorporeted, Huntsville, USA.
[5] Smith, M.J.S. (1998) *ASICs… the course* based on *ASICs… the book*, ISBN 0-201-50022-1, Addison Wesley Longman
[6]www.verilog.net
[7]ww.vhdl.org