A Comparative analysis of Verilog HDL over VHDL

Gurprasad Srivastava
Final Year B. Sc. Engineering, Faculty of Engineering, Dayalbagh Educational Institute, Agra
gurprasad@email.com

Introduction

Hardware description languages (HDLs) were developed as a means to provide varying levels of abstraction to designers. Integrated circuits are too complex for an engineer to create by specifying the individual transistors and wires. HDLs allow the performance to be described at a high level and simulation synthesis programs can then take the language and generate the gate level description. The search for the perfect HDL is like the search for the perfect car which probably doesn't exist. With HDLs, it is no different. The decision of which language to choose is based on a number of important baseline requirements (factors), particularly for the first time user. For the most part, the use of an HDL based design strategy should improve the first time user's productivity. 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 specialized design tools and 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 and that EDA tools vendors provide tools that provide an environment allowing both languages to be used in unison. For example, a designer might have a model of a USB interface written in VHDL, but wants to use it in a design with macros written in Verilog.

History

VHDL became IEEE standard 1076 in 1987. It was updated in 1993 and is known today as "IEEE standard 1076 1993". The Verilog HDL has been used far longer than VHDL and has been used extensively since it was launched by Gateway in 1983. Cadence bought Gateway in 1989 and opened Verilog to the public domain in 1990. It became IEEE standard 1364 in December 1995.

Verilog HDL The Designer's Choice

Lexical Elements

The language is case sensitive and all the keywords are lower case. White space, namely, spaces, tabs and new-lines are ignored. Verilog has two types of comments:

- One line comments start with // and end at the end of the line.
- Multi-line comments start with /* and end with */

Variable names have to start with an alphabetic character or underscore followed by alphanumeric or underscore characters. The only exception to this is the system tasks and functions which start with a dollar sign. Escaped identifiers (identifier whose first character is a backslash (\ )) permit non alphanumeric characters in Verilog name. The escaped name includes all the characters following the backslash until the first white space character.

Integer Literals

Integer literals can have underscores embedded in them for improved readability. For example,

Binary literal	2'b1Z
Octal literal	2'O17
Decimal literal	9 or 'd9
Hexadecimal literal	3'h189
Decimal literal	24_000

Data Types

The values z and Z stand for high impedance and x and X stand for

uninitialized variables or nets with conflicting drivers. String symbols are enclosed within double quotes ("string").and cannot span multiple lines. Real number literals can be either in fixed notation or in scientific notation.

```
real a, b, c ; // a,b,c to be real
integer j, k ; // integer variable
integer i[1:32] ; // array of
integer variables
time newtime ;
/* time and integer are similar in
functionality, time is an unsigned
64-bit used for time variables */
reg [8*14:1] string ; /* This
defines a vector with range
[msb_expr: lsb_expr] */
initial begin
a = 0.5 ; // same as 5.0e-1. real
variable
b = 1.2E12 ;
c = 26.19_60_e-11 ; /* _'s are
used for readability */
string = " string example " ;
newtime =$time;
end
```

Registers and Nets

A register stores its value from one assignment to the next and is used to model data storage elements.

```
reg [5:0] din ; /* a 6-bit
vector register: individual
bits din[5],.... din[0] */
```

Nets correspond to physical wires that connect instances. The default range of a wire or reg is one bit. Nets do not store values and have to be continuously driven. If a net has multiple drivers (for example two gate outputs are tied together), then the net value is resolved according to its type.

wire	**tri**
wand	**triand**
wor	**trior**
tri0	**tri1**
supply0	**supply1**
trireg	

For a wire, if all the drivers have the same value then the wire resolves to this value. If all the drivers except one have a value of z then the wire resolves to the non z value. If two or more non z drivers have different drive strength, then the wire resolves to the stronger driver. If two drivers of equal strength have different values, then the wire resolves to x. A **trireg** net behaves like a wire except that when all the drivers of the net are in high impedance (z) state, then the net retains its last driven value. **trireg's** are used to model capacitive networks.

A wand net or **triand** net operates as a wired and(**wand**), and a **wor** net or **trior** net operates as a wired or (**wor**), **tri0** and **tri1** nets model nets with resistive pulldown or pullup devices on them. When a **tri0** net is not driven, then its value is 0. When a **tri1** net is not driven, then its value is 1. **supply0** and **supply1** model nets that are connected to the ground or power supply. Memories are declared using register statements with the address range specified as in the following example, The keyword **scalared** allows access to bits and parts of a bus and vectored allows the vector to be modified only collectively.

```
wire vectored [5:0] neta;
/* a 6-bit vectored net */
tri1 vectored [5:0] netb;
/* a 6-bit vectored tri1 */
```

Compiler Directives

Verilog has compiler directives which affect the processing of the input files. The directives start with a grave accent (`) followed by some keyword. A directive takes effect from the point that it appears in the file until either the end of all the files, or until another directive that cancels the effect of the first one is encountered. For example,

```
`define OPCODEADD 00010
```

This defines a macro named OPCODEADD. When the text `OPCODEADD appears in the text, then it is replaced by 00010. Verilog macros are simple text substitutions and do not permit arguments.

```
`ifdef SYNTH <Verilog code>
`endif
```

If "SYNTH" is a defined macro, then the Verilog code until `endif is inserted for the next processing phase. If "SYNTH" is not defined macro then the code is discarded.

```
`include <Verilog file>
```

The code in <Verilog file> is inserted for the next processing phase. Other standard compiler directives are listed below:

```
`resetall - resets all
compiler directives to default
values
`define - text-macro
substitution
`timescale 1ns / 10ps -
specifies time unit/precision
`ifdef, `else, `endif -
conditional compilation
`include - file inclusion
`signed, `unsigned - operator
selection (OVI 2.0 only)
`celldefine, `endcelldefine -
library modules
`default_nettype wire -
default net types
`unconnected_drive
pull0|pull1,
`nounconnected_drive - pullup
or down unconnected ports
`protect and `endprotect -
encryption capability
`protected and `endprotected -
encryption capability
`expand_vectornets,
`noexpand_vectornets,
`autoexpand_vectornets -
vector expansion options
`remove_gatename,
`noremove_gatenames
- remove gate names for more
than one instance
`remove_netname,
`noremove_netnames
- remove net names for more
than one instance
```

System Tasks and Functions

System tasks are tool specific tasks and functions.

```
$display( "Example of using
function"); /* display to screen
*/
$monitor($time, "a=%b, clk = %b,
add=%h",a,clk,add); // monitor
signals
$setuphold( posedge clk, datain,
setup, hold); // setup and hold
checks
```

A list of standard system tasks and functions are listed below:

```
$display, $write - utility to
display information
$fdisplay, $fwrite - write to
file
$strobe, $fstrobe -
display/write simulation data
$monitor, $fmonitor - monitor,
display/write information to
file
$time, $realtime - current
simulation time
$finish - exit the simulator
$stop - stop the simulator
$setup - setup timing check
$hold, $width- hold/width timing
check
$setuphold - combines hold and
setup
$readmemb/$readmemh - read
stimulus patterns into memory
$sreadmemb/$sreadmemh - load
data into memory
$getpattern - fast processing of
stimulus patterns
$history - print command history
$save, $restart, $incsave
- saving, restarting,
incremental saving
$scale - scaling timeunits from
another module
$scope - descend to a particular
hierarchy level
$showscopes - complete list of
named blocks, tasks, modules...
$showvars - show variables at
scope
```

Reserved Keywords

The following lists the reserved words of Verilog hardware description language, as of OVI LRM 2.0.

```
and   always  assign   attribute
begin buf     bufif0   bufif1
case  cmos    deassign default
defparam      disable  else
endattribute  end      endcase
endfunction   endprimitive
endmodule     endtable endtask
event         for      force
forever  fork  function  highz0
highz1   if    initial   inout
input    integer join    large
medium   module  nand   negedge
nor      not     notif0  notif1
nmos     or     output parameter
pmos     posedge  primitive
pulldown   pullup   pull0  pull1
rcmos     reg    release  repeat
rnmos     rpmos  rtran  rtranif0
rtranif1  scalared small specify
specparam  strong0  strong1
supply0   supply1   table  task
```

```
tran tranif0 tranif1 time  tri
triand  trior  trireg  tri0  tri1
vectored  wait  wand  weak0  weak1
while wire wor
```

Structures and Hierarchy

Hierarchical HDL structures are achieved by defining modules and instantiating modules. Nested module definitions (i.e. one module definition within another) are not permitted.

Module Declarations

The module name must be unique and no other module or primitive can have the same name. The port list is optional. A module without a port list or with an empty port list is typically a top level module. A macromodule is a module with a flattened hierarchy and is used by some simulators for efficiency.

```
module dff (q,qb,clk,d,rst);
input clk,d,rst ; // input
signals
output q,qb ; // output
definition
//inout for bidirectionals
// Net type declarations
wire dl,dbl ;
// parameter value assignment
paramter delay1 = 3,
delay2 = delay1 + 1; // delay2
// shows parameter dependance
/* Hierarchy primitive
instantiation, port
connection in this section is by
ordered list */
nand #delay1 n1(cf,dl,cbf),
n2(cbf,clk,cf,rst);
nand #delay2 n3(dl,d,dbl,rst),
n4(dbl,dl,clk,cbf),
n5(q,cbf,qb),
n6(qb,dbl,q,rst);
/***** for debuging model
initial begin
#500 force dff_lab.rst = 1 ;
#550 release dff_lab.rst;
// upward path referencing
end ********/
endmodule
```

Expressions and Operators

Arithmetic and logical operators are used to build expressions. Expressions perform operation on one or more operands, the operands being vectored or scalared nets, registers, bit-selects, part selects, function calls or concatenations thereof.

• Unary Expression
<operator> <operand>

a = ! b ;

• Binary and Other Expressions
<operand> <operator> <operand>

if (a < b) // if (<expression>)
{c,d} = a + b ;
// concatenate and add operator

• Parentheses can be used to change the precedence of operators. For example,
((a+b) * c)

• All operators associate left to right, except for the ternary operator "?:" which associates from right to left.

Verilog or VDHL

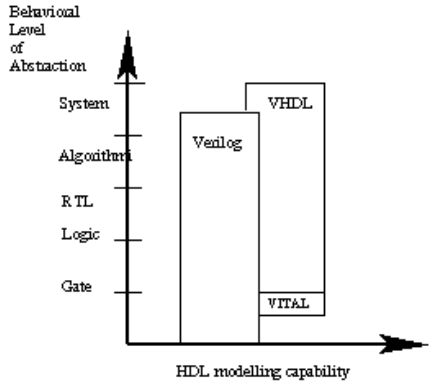
There are two aspects to modelling hardware that any HDL facilitates; True Abstract Behaviour and Hardware Structure. This means modelled hardware behaviour is not prejudiced by structural or design aspects of hardware intent and that hardware structure is capable of being modelled irrespective of the design's behaviour. Here we compare the two HDLs for their various similarities and differences.

Capability

Hardware structure can be modelled equally effectively in both VHDL and Verilog. When modelling abstract hardware, the capability of VHDL can sometimes only be achieved in Verilog when using the PLI. The choice of which to use is not therefore based solely on technical capability but on:

- personal preferences
- EDA tool availability
- commercial, business and marketing issues

The modelling constructs of VHDL and Verilog cover a slightly different spectrum across the levels of behavioural abstraction.



Comparative capabilities of Verilog & VHDL

Compilation

VHDL - Multiple design–units which reside in the same system file may be separately compiled if so desired. 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.

Verilog - 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.

Data Types

VHDL - A multitude of language or user defined data types can be used. This may mean dedicated conversion functions are needed to convert objects from one type to another. The choice of which data types to use should be considered wisely, especially enumerated (abstract) data types. This will make models easier to write, clearer to read and avoid unnecessary conversion functions that can clutter the code. VHDL may be preferred because it allows a multitude of language or user defined data types to be used.

Verilog - Compared to VHDL, Verilog data types a re very simple, easy to use and very much geared towards modelling hardware structure as opposed to abstract hardware modelling. Unlike VHDL, all data types used in a Verilog model are defined by the Verilog language and not by the user. There are net data types, for example wire, and a register data type called reg. A model with a signal whose type is one of the net data types has a corresponding electrical wire in the implied modelled circuit. Objects that are signals of type reg hold their value over simulation delta cycles and should not be confused with the modelling of a hardware register. Verilog may be preferred because of its simplicity.

Design Reusability

VHDL - Procedures and functions may be placed in a package so that they are avail able to any design-unit that wishes to use them.

Verilog - There is no concept of packages in 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 the `include compiler directive.

Easiest to Learn

Starting with zero knowledge of either language, Verilog is probably the easiest to grasp 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 primary 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.

Forward & back annotation

A spin-off from Verilog is the Standard Delay Format (SDF). This is a general purpose format used to define the timing delays in a circuit. The format provides a bidirectional link between, chip layout tools, and either synthesis or simulation tools, in order to provide more accurate timing representations. The SDF format is now an industry standard in it's own right.

High Level Constructs

VHDL - There are more constructs and features for high-level modelling 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, for example, bit width.

All these language statements are useful in synthesizable models.

Verilog - Except for being able to parameterize models by overloading parameter constants, there is no equivalent to the high-level VHDL modelling statements in Verilog.

Language Extensions

The use of language extensions will make a model non standard and most likely not portable across other design tools. However, sometimes they are necessary in order to achieve the desired results.

VHDL - Has an attribute called 'foreign that allows architectures and subprograms to be modelled in another language.

Verilog - The Programming Language Interface (PLI) is an interface mechanism between Verilog models and Verilog software tools. For example, a designer, or more likely, a Verilog tool vendor, can specify user defined tasks or functions in the C programming language, and then call them from the Verilog source description. Use of such tasks or functions make a Verilog model non-standard and so may not be usable by other Verilog tools. Their use is not recommended.

Libraries

VHDL - A library is a store for compiled entities, architectures, packages and configurations; useful for managing multiple design projects.

Verilog - There is no concept of a library in Verilog. This is due to it's origins as an interpretive language.

Low Level Constructs

VHDL - Simple two input logical operators are built into the language, they are: NOT, AND, OR, NAND, NOR, XOR and XNOR. Any timing must be separately specified using the after clause. Separate constructs defined under the VITAL language must be used to define the cell primitives of ASIC and FPGA libraries.

Verilog - The Verilog language was originally developed with gate level modelling in mind, and so has very good constructs for modelling at this level and for modelling the cell primitives of ASIC and FPGA libraries. Examples include User Defined Primitive s (UDP), truth tables and the specify block for specifying timing delays across a module.

Managing large designs

VHDL - Configuration, generate, generic and package statements all help manage large design structures.

Verilog - There are no statements in Verilog that help manage large designs.

Operators

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.

Parameterizable models

VHDL - A specific bit width model can be instantiated from a generic n-bit model using the generic statement. The generic model will not synthesize until it is instantiated and the value of the generic given.

Verilog - A specific width model can be instantiated from a generic n-bit model using overloaded parameter values. The generic model must have a default parameter value defined. This means two things. In the absence of an overloaded value being specified, it will still synthesize, but will use the specified default parameter value. Also, it does not need to be instantiated with an overloaded parameter value specified, before it will synthesize.

Procedures and tasks

VHDL allows concurrent procedure calls; Verilog does not allow concurrent task calls.

Readability

This is more a matter of coding style and experience than language feature. VHDL is a concise and verbose language; its roots are based on ADA. Verilog is more like C because its constructs are based approximately 50% on C and 50% on ADA. For this reason an existing C programmer may prefer Verilog over VHDL. Although an existing programmer of both C and ADA may find the mix of constructs somewhat confusing at first. Whatever HDL is used, when writing or reading an HDL model to be synthesized it is important to think about hardware intent.

Structural replication

VHDL - The *generate* statement replicates a number of instances of the same design-unit or some sub part of a design, and connects it appropriately.

Verilog - There is no equivalent to the *generate* statement in Verilog.

Test harnesses

Designers typically spend about 50% of their time writing synthesizable models and the other 50% writing a test harness to verify the synthesizable models. Test harnesses are not restricted to the synthesizable subset and so are free to use the full potential of the language. VHDL has generic and configuration statements that are useful in test harnesses, which are not found in Verilog.

Verboseness

VHDL - Because VHDL is a very strongly typed language models must be coded precisely with defined and matching data types. This may be considered an advantage or disadvantage. However, it does mean models are often more verbose, and the code often longer, than its Verilog equivalent.

Verilog - Signals representing objects of different bits widths may be assigned to each other. The signal representing the smaller number of bits is automatically padded out to that of the larger number of bits, and is independent of whether it is the assigned signal or not. Unused bits will be automatically optimized away during the synthesis process. This has the advantage of not needing to model quite so explicitly as in VHDL, but does mean unintended modelling errors will not be identified by an analyzer.

Conclusion

To summarize, VHDL was developed (for the US DOD) to provide a consistent modelling language for the documentation of digital hardware designs. The language was never intended to be used to do actual design.

However, to maintain a supposed competitive advantage, individual EDA companies exerted considerable influence, resources and dollars to force the language to become a design language. These same EDA companies implemented their own semi-unique versions of the language at different stages during its development. This means VHDL models that were developed on one system, may not run on a different system.

The language is difficult to learn and even more difficult to use. It is extremely verbose, especially at the gate level, when timing information is specific and considerable. VHDL's verbosity causes severe memory problems when trying to simulate medium to large designs. ASIC vendors have been very reluctant to provide VHDL gate level libraries that include full timing because of the size of the models and the abnormally long simulation times associated with validating a relatively simple design.

The framers of VHDL were driven by the US DOD, which has no material interest in design productivity. VHDL's complex syntax interferes with design productivity and does not offer any strategic advantage that would improve the quality of the design. This essentially undermines the basic strength of VHDL, productivity achieved via a methodology based on top-down-design.

Verilog HDL has been developed and will continue to evolve to address the needs and commercial applications of the design community that has made it the most successful language in use today. The design community has invested almost 20 billion dollars in Verilog HDL and related tools over the last 8 years. The ability to address higher level language constructs are well supported in the language, along with its rock solid structural (gate and switch level) strengths. As long as designers and their companies have to get high quality innovative products to market in the time sensitive world in which we all compete, Verilog HDL will continue to be the dominant solution. Almost every major computer manufacturer, system developer, ASIC and semiconductor manufacturer uses Verilog HDL as their modelling language.

For the first time HDL user the selection of Verilog HDL as your modelling language will be a very wise decision. It will mean there are a number of tools available from schematic entry to synthesis to simulation at various price ranges and on numerous platforms from PCs to mainframes. There are also numerous libraries available from a variety of sources that support full timing based models with all the necessary delay functionality required to meet any critical design needs. There is also a vast resource of Verilog HDL engineering talent that has had experience using the language for practical commercial design to provide critical assistance if it becomes necessary. There are many HDLs you can choose from but only one that has proven time and time again that it is the only choice for real designs.

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. The choice of HDL is shown not to be based on technical capability, but on: personal preferences, EDA tool availability and commercial, business and marketing issues.

Reference

[1] VHDL & Verilog Compared & Contrasted Plus Modeled Example Written in VHDL, Verilog and C.

Douglas J. Smith

[2] Verilog HDL vs. VHDL for the First Time User.

Bill Fuchs

[3] Quick Reference for Verilog HDL.

Rajeev Madhavan

[4] Verilog-A Reference Manual

Agilent Technologies

[5] Verilog Hardware Description Language Reference Manual.

Open Verilog International