Programmable Logic Design

Grzegorz Budzyń

Lecture 12: VHDL vs Verilog

Choose yourself and new technologies

HUMAN CAPITAL
HUMAN – BEST INVESTMENT!

Wrocław University of Technology

EUROPEAN SOCIAL FUND

Project co-financed from the EU European Social Fund

Plan

- Introduction
- Verilog in brief
- VHDL/Verilog comparison
- Examples
- Summary

Introduction

HUMAN CAPITAL
HUMAN – BEST INVESTMENT!
Wrocław University of Technology
EUROPEAN SOCIAL FUND
Project co-financed from the EU European Social Fund

Introduction

- At present there are two industry standard hardware description languages, VHDL and Verilog.
- The complexity of ASIC and FPGA designs has meant an increase in the number of specific tools and 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.

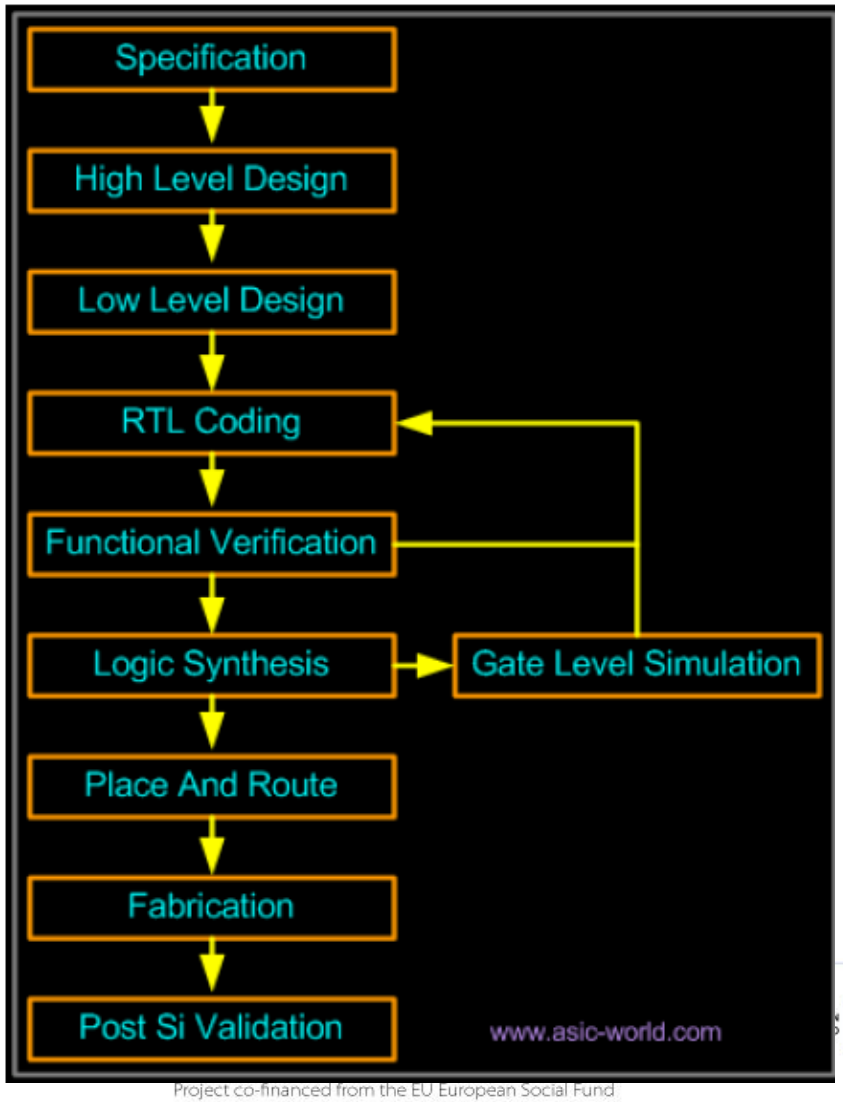
Introduction

- VHDL (Very high speed integrated circuit Hardware Description Language) became IEEE standard 1076 in 1987.
- The Verilog hardware description language 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.
- Verilog became IEEE standard 1364 in December 1995

Verilog in brief

HUMAN CAPITAL
HUMAN – BEST INVESTMENT!
Wrocław University of Technology
EUROPEAN SOCIAL FUND
Project co-financed from the EU European Social Fund

Verilog – design flow



Verilog – abstraction levels

- Verilog supports three main abstraction levels:
  - Behavioral level – a system is described by concurrent algorithm
  - Register-transfer level – a system is characterised by by operations and transfer of data between registers according to an explicit clock
  - Gate level – a system is described by logical links and their timing characteristics

Verilog – „Hello world”

```
////////////////////////////////////////////
module HelloWorld;

initial begin
   $display("Hello World");
   #10 $finish;
end

endmodule



////////////////////////////////////////////
module test(
    input clk,
    input rst,
    output q
    );


endmodule
```

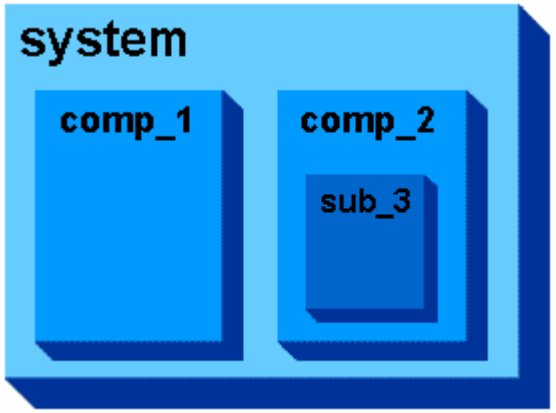
Verilog – 8-bit counter example

```
//----------------------
module counter(
clk,
rst,
cnt_out
);
//End of port list
//--------Input ports-------
input clk;
input rst;
//-------Output ports-------
output [7:0] cnt_out;
//----Input port types------
//by rule all the input ports should be wires
wire clk;
wire rst;
//----Output port types-----
//output port types can be a wire or a storage element (reg)
reg [7:0] cnt_out;

//----Code starts here------
always @ (posedge clk)
begin : COUNTER //block name
   if (reset == 1'b1) begin
      cnt <= #1 8b'00000000;
   end
   else
      cnt <= #1 cnt + 1;
   end
end

endmodule
```

Basic constructs

Verilog - hierarchy

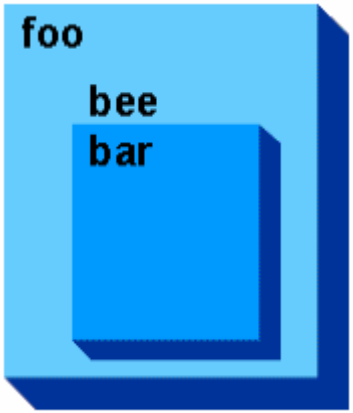
- Verilog structures which build the hierarchy are:
  - modules
  - ports
- A module is the basic unit of the model, and it may be composed of **instances** of other modules



Verilog - hierarchy

- the top level module is not instantiated by any other module
- Example:

```
module foo;
   bar bee (port1, port2);
endmodule
module bar (port1, port2);
   ...
endmodule
```



Verilog

- Port types in Verilog:
  - Input
  - Output
  - Inout
- Matching ports by names:

```
foo f1 (.bidi(bus), .out1(sink1), .in1(source1));
```

versus normal way

```
foo f1 (source1, , sink1, , bus);
```

Verilog - modules

- Verilog models are made up of modules
- Modules are made of different types of components:
  - Parameters
  - Nets
  - Registers
  - Primitives and Instances
  - Continuous Assignments
  - Procedural Blocks
  - Task/Function definitions

Verilog - parameters

- *Parameters* are constants whose values are determined at compile-time
- They are defined with the parameter statement:

```
parameter identifier = expression;
```

Example:

```
parameter width = 8, msb = 7, lsb = 0;
```

Verilog - nets

- Nets are the things that connect model components together – like signals in VHDL
- Nets are declared in statements like this:

net_type [range] [delay3] list_of_net_identifiers

- Example:

wire w1, w2;

tri [31:0] bus32;

Verilog – types of nets

Net Data Type	Functionality
wire, tri	Interconnecting wire – no special resolution function
wor, trior	Wired outputs OR together (models ECL)
wand,triand	Wired outputs AND together (models open–collector)
tri0,tri1	Net pulls–down or pulls–up when not driven
supply0,suppy1	Net has a constant logic 0 or logic 1 (supply strength)
trireg	

- Each net type has functionality that is used to model different types of hardware such as CMOS, NMOS, TTL etc

Verilog – net drivers

- Nets are driven by net drivers.
- Drivers may be:
  - output port of a primitive instance
  - output port of a module instance
  - left-hand side of a continuous assignment
- There may be more than one driver on a net
- If there is more than one driver, the value of the net is determined by a **built-in** resolution function

Verilog - registers

- *Registers* are storage elements
- Values are stored in registers in procedural assignment statements
- Registers can be used as the source for a primitive or module instance (i.e. registers can be connected to input ports), but they cannot be driven in the same way a net can
- Examples:
  - reg r1, r2;
    reg [31:0] bus32;
    integer i;

Verilog - registers

- There are four types of registers:
  - **Reg:**
    - This is the generic register data type. A reg declaration can specify registers which are 1 bit wide to 1 million bits wide
  - **Integer**
    - Integers are 32 bit signed values
  - **Time**
    - Registers declared with the time keyword are 64-bit unsigned integers
  - **Real (and Realtime)**
    - Real registers are 64-bit IEEE floating point

Verilog - memories

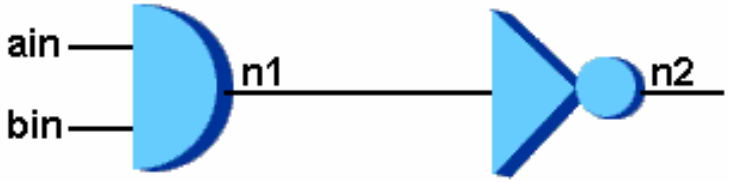
- Verilog allows arrays of registers, called memories
- Memories are static, single-dimension arrays
- The format of a memory declaration is:
  - reg [range] identifier range ;
- Example:
  - reg [0:31] temp, mem[1:1024];
  - ...
  - temp = mem[10]; --extract 10th element
  - bit = temp[3]; --extarct 3rd bit

Verilog - primitives

- Primitives are pre-defined module types
- The Verilog primitives are sometimes called gates, because for the most part, they are simple logical primitives
- Examples:
  - And, nand, or, nor, xor, xnor
  - Buf, not
  - Pullup, pulldown
  - bufif0, notif0

Verilog - primitives

- Examples:




```
module test;
   wire n1, n2;
   reg ain, bin;

   and and_prim(n1, ain, bin);
   not not_prim(n2, n1);
endmodule
```

Procedural blocks

Verilog - Continuous assignments

- Continuous assignments are known as data flow statements
- They describe how data moves from one place, either a net or register, to another
- They are usually thought of as representing combinational logic
- Examples:

```
assign w1 = w2 & w3;
assign (strong1, pull0) mynet = enable;
```

Verilog - Procedural blocks

- Procedural blocks are the part of the language which represents sequential behavior
- A module can have as many procedural blocks as necessary
- These blocks are sequences of executable statements
- The statements in each block are executed sequentially, but the blocks themselves are concurrent and asynchronous to other blocks

Verilog - Procedural blocks

- There are two types of procedural blocks, *initial blocks* and *always blocks*
- All initial and always blocks contain a single statement, which may be a compound statement, e.g.:

```
initial
    begin statement1 ; statement2 ; ... end
```

Verilog - Initial blocks

- All initial blocks begin at time 0 and execute the initial statement
- Because the statement may be a compound statement, this may entail executing lots of statements
- An initial block may cause activity to occur throughout the entire simulation of the model
- When the initial statement finishes execution, the initial block terminates

Verilog - Initial blocks

- Examples:

```
initial x = 0; // a simple initialization

initial begin
      x = 1;        // an initialization
      y = f(x);
#1 x = 0;         // a value change 1 time unit later
      y = f(x);
end
```

Verilog – Always block

- Always blocks also begin at time 0
- The only difference between an always block and an initial block is that when the always statement finishes execution, it starts executing again

Tasks and functions

Verilog – Tasks/functions

- Tasks and functions are declared within modules
- Tasks may only be used in procedural blocks
- A task invocation is a statement by itself. It may not be used as an operand in an expression

- Functions are used as operands in expressions
- A function may be used in either a procedural block or a continuous assignment, or indeed, any place where an expression may appear

Verilog – Tasks

- Tasks may have zero or more arguments, and they may be input, output, or inout arguments
- Time can elapse during the execution of a task, according to time and event controls in the task definition
- Exmaple:

```
task do_read;
   begin
     adbus_reg = addr; // put address out
   end
endtask
```

Verilog – Functions

- In contrast to tasks, functions must execute in a single instant of simulated time
- That is, not time or delay controls are allowed in a function
- Function arguments are also restricted to inputs only.
- Output and inout arguments are not allowed.
- The output of a function is indicated by an assignment to the function name

Verilog – Functions

- Example:

```
function [15:0] relocate;
    input [11:0] addr;
    input [3:0] relocation_factor;
    begin
      relocate = addr + (relocation_factor<<12);
      count = count + 1; // how many have we done end
    endfunction

    assign absolute_address = relocate(relative_address,
    rf);
```

VHDL/Verilog comparison

Capability

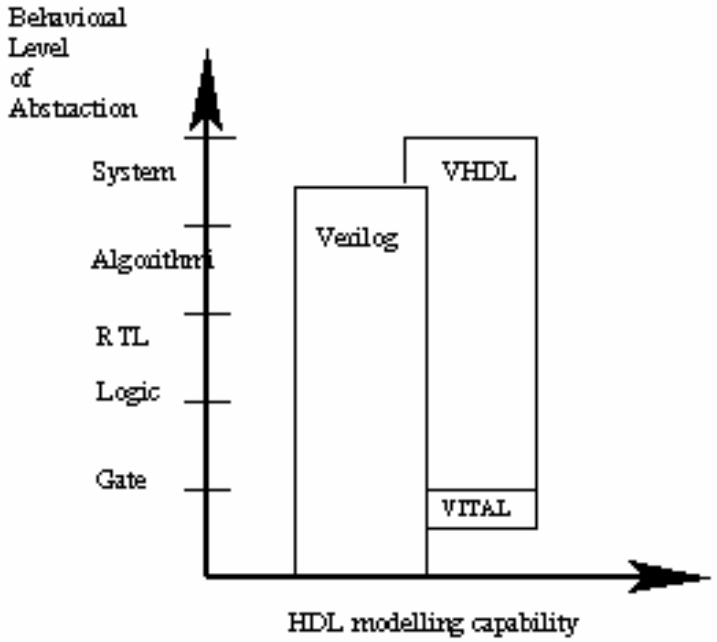
- VHDL – like Pascal or Ada programming languages
- Verilog – like C programming language
- It is important to remember that both are **Hardware Description Languages** and not programming languages
- For synthesis only a subset of languages is used

Capability

- 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.
- 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

Capability

- The modeling constructs of VHDL and Verilog cover a slightly different spectrum across the levels of behavioral abstraction



Source: [1]

Compilation

- VHDL:
  - Multiple design-units (entity/architecture pairs), that reside in the same system file, may be separately compiled if so desired.
  - It is good design practice to keep each design unit in it's own system file in which case separate compilation should not be an issue.

Compilation

- Verilog:
  - The Verilog language is still rooted in it's native interpretative mode.
  - Compilation is a means of speeding up simulation, but has not changed the original nature of the language.
  - The 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.
  - VHDL may be preferred because it allows a multitude of language or user defined data types to be used.

Data types

- Verilog:
  - Compared to VHDL, Verilog data types a very simple, easy to use and very much geared towards modeling hardware structure as opposed to abstract hardware modeling.
  - 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 modeled circuit.
  - Verilog may be preferred because of it's simplicity.

Design reusability

- VHDL:
  - Procedures and functions may be placed in a package so that they are available 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.

Ease of learning

- Starting with zero knowledge of either language, Verilog is probably the easiest to grasp and understand.
- 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, specially those with large hierarchical structures

High level constructs

- VHDL:
  - There are more constructs and features for high-level modeling 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.

High level constructs

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

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.

Low level constructs

- Verilog:
  - The Verilog language was originally developed with gate level modeling in mind, and so 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.

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.

Procedures and tasks

- VHDL:
  - concurrent procedure calls are allowed
- Verilog:
  - concurrent procedure calls are not allowed

Readability

- This is more a matter of coding style and experience than language feature.
- VHDL is a concise and verbose language;
- Verilog is more like C because it's constructs are based approximately 50% on C and 50% on Ada.
- For this reason an existing C programmer may prefer Verilog over VHDL.
- 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.

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.
  - It does mean models are often more verbose, and the code often longer, than it's Verilog equivalent.

Verboseness

- 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 modeling errors will not be identified by an analyzer.

Examples

Binary up counter

- VHDL:

```
process (clock)
begin
    if clock='1' and clock'event then
        counter <= counter + 1;
    end if;
end process;
```

Binary up counter

- Verilog:

```
reg [upper:0] counter;

    always @(posedge clock)
       counter <= counter + 1;
```

D FilpFlop

- VHDL:

```
process (<clock>)
begin
    if <clock>'event and <clock>='1'
  then
        <output> <= <input>;
    end if;
end process;
```

D FilpFlop

- Verilog:

```
always @(posedge <clock>) begin
    <reg> <= <signal>;
  end
```

Synchronous multiplier

- VHDL:

```
process (<clock>)
begin
    if <clock>='1' and <clock>'event
  then
        <output> <= <input1> * <input2>;
    end if;
end process;
```

Synchronous multiplier

- Verilog:

```
wire [17:0] <a_input>;
  wire [17:0] <b_input>;
  reg  [35:0] <product>;

  always @(posedge <clock>)
     <product> <= <a_input> *
<b_input>;
```

Summary

	VHDL	Verilog
Strong typing	Yes	No
User-defined types	Yes	No
Dynamic memory allocation	Yes	No
Physical types	Yes	No
Enumerated types	Yes	No
Records/structs	Yes	No

	VHDL	Verilog
Bit (vector) / integer equivalence	Partial (by libraries)	Yes
Subprograms	Yes	Yes
Separate packaging	Yes Packages	Yes Include files
Gate level modeling	Yes VITAL	Yes Builtin primitives

	VHDL	Verilog
Conditional statements	Yes • If-then-else/elsif (priority) • Case (mux) • Selected assign (mux) • Conditional assign (priority) • No “don’t care” matching capability	Yes • if-else (priority) • case (mux) • casex (mux) • ?: (conditional used in concurrent assignments)

	VHDL	Verilog
Iteration	Yes • Loop • while-loop • for-loop • exit • next	Yes • repeat • for • while

Thank you for your attention

HUMAN CAPITAL
HUMAN – BEST INVESTMENT!
Wrocław University of Technology
EUROPEAN SOCIAL FUND
Project co-financed from the EU European Social Fund

References

[1] Douglas J. Smith, „VHDL & Verilog Compared & Contrasted Plus Modeled Example Written in VHDL, Verilog and C”

[2] http://www.stanford.edu/class/ee183/handouts_win2003/VerilogQuickRef.pdf

[3] www.asic-world.com

[4] http://www.ece.umd.edu/courses/enee359a/verilog_tutorial.pdf