VHDL & Verilog Compared & Contrasted - Plus Modeled Example Written in VHDL, Verilog and C.

Douglas J. Smith
VeriBest Incorporated
One Madison Industrial Estate, Huntsville, AL 35894-0001, USA
e-mail: djsmith@ingr.com

Abstract

This tutorial is in two parts. The first part takes an unbiased view of VHDL and Verilog by comparing their similarities and contrasting their diffrences. The second part contains a worked example of a model that computes the Greatest Common Divisor (GCD) of two numbers. The GCD is modeled at the algorithmic level in VHDL, Verilog and for comparison purposes, C. It is then shown modeled at the RTL in VHDL and Verilog.

1. Introduction

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 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 PCI bus interface written in VHDL, but wants to use it in a design with macros written in Verilog.

2. Background

VHDL (Very high speed intgrated circuit Hardware Description Language) became IEEE standard 1076 in 1987. It was updated in 1993 and is known today as “IEEE standard 1076 1993”. The Verilog hardware description language has been used far longer than VHDL and has been used extensively since it was lauched 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.

There are two aspects to modeling hardware that any hardware description language facilitates; true abstract behavior and hardware structure. This means modeled hardware behavior is not prejudiced by structural or design aspects of hardware intent and that hardware structure is capable of being modeled irrespective of the design's behavior.

3. VHDL/Verilog compared & contrasted

This section compares and contrasts individual aspects of the two languages; they are listed in alphabetical order.

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

The modeling constructs of VHDL and Verilog cover a slightly different spectrum across the levels of behavioral abstraction; see Figure 1.

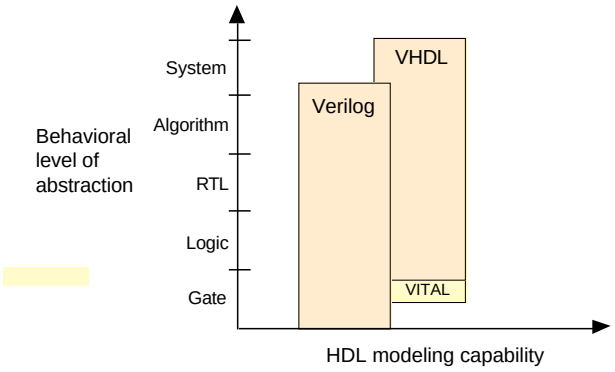

Figure 1. HDL modeling capability

Compilation

VHDL. Multiple *design-units* (**entity**/**architecture** pairs), that 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 it’s own system file in which case separate compilation should not be an issue.

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. 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 are very simple, easy to use and very much geared towards modeling hardware structure as opposed to abstract hardware modeling. Unlike VHDL,

33rd Design Automation Conference ®

DAC 96 - 06/96 Las Vegas, NV, USA

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. Objects, that is signals, of type **reg** hold their value over simulation delta cycles and should not be confused with the modeling of a hardware register. 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 `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, specially those with large hierarchical structures.

Forward and 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 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.

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.

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 modeled 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 nonstandard 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 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.

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 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. 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, that 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 it's 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 modeling errors will not be identified by an analyzer.

4. Greatest Common Divisor

Models of a greatest common divisor circuit is posed as problem and solution exercise. A model written in C is included in addition to VHDL and Verilog for comparison purposes.

4.1 Problem

The problem consists of three parts:

1. Design three algorithmic level models of an algorithm that finds the Greatest Common Divisor (GCD) of two numbers in the software programming language, C, and the two hardware description languages, VHDL and Verilog. Use common test data files to test the algorithm where practically possible. Neither the VHDL nor Verilog models need contain timing. All three models should automatically indicate a pass or fail condition.
2. Model the GCD algorithm at the RTL level for synthesis in both VHDL and Verilog. The model must be generic so that it can be instantiated with different bit widths. A `Load` signal should indicate when input data is valid, and a signal called `Done`, should be provided to signify when valid output data is available. The generic model should be verified with 8-bit bus signals.
3. Write VHDL and Verilog test harnesses for the two models that: 1) use the same test data files used by the algorithmic level models, and 2), instantiates both the RTL and synthesized gate level models so that they are simulated and tested at the same time.

4.2 Solution

The solution is broken into three parts corresponding to those of the problem.

1. Designing algorithmic level models in C, VHDL and Verilog

The algorithm used to find the greatest common divisor between two numbers is shown in Figure 2.

It works by continually subtracting the smaller of the two numbers, A or B, from the largest until such point the smallest number becomes equal to zero. It does this by continually subtracting B from A while A is greater than B, and then swapping A and B around when A becomes less than B so that the new value of B can once again be continually subtracted from A. This process continues until B becomes zero.

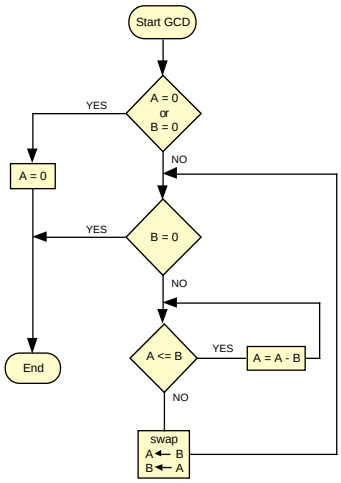

Figure 2 GCD Algorithm

C model

The C model first declares integer values for the two inputs `A` and `B`, the computed output of the algorithm `Y`, and the reference output `Y_Ref`. Integer `Y_Ref` is the expected GCD result and used to compare with the computed result from the algorithm. The integer `Swap` is also declared and used in the algorithm to swap the two inputs `A` and `B`. A final integer, `Passed`, is used to indicate a pass (1) or fail (0) condition.

A file pointer (`file_pointer`) is defined in order to access the test data file "`gcd_test_data.txt`". It is opened for read mode only. Integer `Passed` is initially set to 1 and only set to 0 if the algorithm fails.

Reading test data file. The test data file contains three numbers on each line corresponding to values of `A`, `B` and `Y_Ref` respectively. A **while** loop is used to: 1) read each line of the test data file, 2) assign the three values to `A`, `B` and `Y_Ref` respectively, 3) use `A` and `B` to compute the GCD output `Y`, and 4) compare `Y` with `Y_Ref`. This **while** loop continues while there is test data in the test data file.

Algorithm implementation. The initial **if** statement is an extra check that both `A` and `B` are not zero. The algorithm is then modeled using two **while** statements. The first, outer-most, **while** statement checks to see if B has reached zero; if it has the GCD has been found. The second, inner-most, **while** statement checks to see if `A` is greater than or equal to `B`; if it is, it continually subtracts `A` from `B` and puts the result back in `A`. When `A`|Ucomes less than `B` the inner most **while** loop completes, `A` and `B` are swapped using `Swap`, and the outer most **while** statement rechecks `B` to see if it has reached zero.

Testing the result. The algorithm is tested using an **if** statement which tests to see if the computed result `Y` is the same as the expected result `Y_Ref`. If they are different an error message is printed to the screen and `Passed` assigned the value 0. Finally, when all tests have

completed and Passed is still equal to 1 a passed message is printed to the screen.

VHDL Model

The VHDL model follows exactly the same principle as defined for the C model above. When reading the integer values from the test date file they must be read and assigned to a variable; they cannot be read and assigned to a signal. As this is an algorithmic level model defined in a single entity it contains no input or ouputs, nor does it contain any internal signals or associated timing. All computations use variables; variables are read from the test data file, the algorithm computes the result and variables are written to a results file.

Verilog Model

The Verilog model also follows the same principle as defined above for the C model. A major difference in this model is that Verilog cannot read decimal integer values from a system file. Data read from a system file must be: 1) read using one of the two language define system tasks, $readmemh or $readmemb and 2) stored in a memory, which has specific width and depth. This limits any read data to being in either hexadecimal or binary format. In this case a separate test data file is used "gcd_test_data_hex.txt" which has the test data specified in hexadecimal format.

file: gcd_test_data.txt

```
21   49   7
25   30   5
19   27   1
40   40   40
250  190  10
5    250  5
```

file: gcd_test_data_hex.txt

```
15  31  7  //  Decimal  21  49  7
19  1E  5  //  Decimal  25  30  5
13  1B  1  //  Decimal  19  27  1
28  28  28 //  Decimal  40  40  40
FA  6E  A  //  Decimal  250 190 10
5   FA  5  //  Decimal  5   250 5
```

C - algorithmic level

```
#include <stdio.h>

main ()
   {
    int A, B, Swap, Y, Y_Ref, Passed;
    FILE *file_pointer;
    file_pointer =
    fopen("gcd_test_data.txt", "r");
    Passed = 1;
    while (!feof(file_pointer))
              {
               /*-------------------------------------*/
               /* Read test data from file */
               /*-------------------------------------*/
               fscanf (file_pointer, "%d %d
               %d\n", &A, &B, &Y_Ref);

               /*-----------------------------------*/
               /* Model GCD algorithm */
               /*-----------------------------------*/
               if (A != 0 && B != 0)
                    {
                     while (B != 0)
                          {
                           while (A >= B)
                                {
                                 A = A - B;
                                }
                           Swap = A;
                           A = B;
                           B = Swap;
                          }
                    }
                else
                    {
                     A = 0;
                    }
               Y = A;

               /*-----------------------------------*/
               /* Test GCD algorithm */
               /*-----------------------------------*/
               if (Y != Y_Ref)
                    {
                     printf ("Error. A=%d B=%d
                           Y=%d Y_Ref=%d\n",
                           A,B,Y,Y_Ref);
                     Passed = 0;
                    }
              }

         if (Passed = 1)
              printf ("GCD algorithm test passed ok\n");
   }
```

VHDL - algorithmic level

```
library STD;
use STD.TEXTIO.all;
entity GCD_ALG is
end entity GCD_ALG;

architecture ALGORITHM of GCD_ALG is
     ------------------------------------------------------
     -- Declare test data file and results file
     ------------------------------------------------------
     file TestDataFile: text open
          read_mode is "gcd_test_data.txt";
     file ResultsFile: text open write_mode is
          "gcd_alg_test_results.txt";
begin
     GCD: process
          variable A,B,Swap,Y,Y_Ref: integer range 0 to 65535;
          variable TestData: line;
          variable BufLine: line;
          variable Passed: bit := '0';
     begin
          while not endfile(TestDataFile) loop
               ---------------------------------------
               -- Read test data from file
               ---------------------------------------
               readline(TestDataFile, TestData);
               read(TestData, A);
               read(TestData, B);
               read(TestData, Y);
               read(TestData, Y_Ref);

               -----------------------------------
               -- Model GCD algorithm
               -----------------------------------
               if (A /= 0 and B /= 0) then
                    while (B /= 0) loop
                         while (A >= B) loop
                              A := A - B;
                         end loop;
                         Swap:= A;
                         A   := B;
                         B   := Swap;
                    end loop;
               else
                    A := 0;
               end if;
               Y := A;

               ---------------------------------
               -- Test GCD algorithm
               ---------------------------------
               if (Y /= Y_Ref) then  -- has failed
                     Passed := '0';
                    write(Bufline, string'("GCD Error: A="));
                    write(Bufline, A);
                    write(Bufline, string'(" B="));
                    write(Bufline, B);
                    write(Bufline, string'(" Y="));
                    write(Bufline, Y);
                    write(Bufline, string'(" Y_Ref="));
```

```
                write(Bufline, Y_Ref);
                writeline(ResultsFile, Bufline);
            end if;
        end loop;
    if (Passed = '1') then  -- has passed
        write(Bufline, string'
                ("GCD algorithm test has passed"));
        writeline(ResultsFile, Bufline);
    end if;
  end process;

end architecture ALGORITHM;
```

Verilog - algorithmic level

```
module GCD_ALG;
    parameter Width = 8;
    reg [Width-1:0] A,B,Y,Y_Ref;
    reg [Width-1:0] A_reg,B_reg,Swap;

    parameter GCD_tests = 6;
    integer N,M;
    reg     Passed,
            FailTime;
    integer SimResults;

    // Declare memory array for test data
    // ------------------------------------------------------
    reg  [Width-1:1] AB_Y_Ref_Arr[1:GCD_tests*3];

    //-----------------------------------
    // Model GCD algorithm
    //-----------------------------------
    always @(A or B)
        begin: GCD
            A_reg = A;
                B_reg = B;
                if (A_reg != 0 && B_reg != 0)
                    while (B_reg != 0)
                        begin
                            while (A_reg >= B_reg)
                                A_reg = A_reg - B_reg;
                            Swap  = A_reg;
                            A_reg = B_reg;
                            B_reg = Swap;
                        end
                else
                    A_reg = 0;
                Y = A_reg;
        end

    //-------------------------------
    // Test GCD algorithm
    //-------------------------------
    initial
        begin
            // Load contents of
            // "gcd_test_data.txt" into array.
            $readmemh("gcd_test_data_hex.txt", AB_Y_Ref_Arr);

            // Open simulation results file
            SimResults = $fopen("gcd.simres");

            Passed = 1;  // Set to 0 if fails
            for (N=1; N<=GCD_tests; N=N+1)
                begin
                    A=AB_Y_Ref_Arr[(N*3)+1];
                        B=AB_Y_Ref_Arr[(N*3)+2];
                    Y_Ref=AB_Y_Ref_Arr[(N*3)+3];
                    #TestPeriod
                    if (Y != Y_Ref)  // has failed
                        begin
                            Passed = 0;
                            $fdisplay (SimResults,
                               "GCD Error: A=%d
                            B=%d Y=%d. Y should be %d",
                            A,B,Y,Y_Ref);
                        end
                end
            if (Passed == 1)     // has passed
                $fdisplay (SimResults,
                    "GCD algorithm test has passed");
                $fclose (SimResults);
                $finish;
        end
endmodule
```

2. Designing RTL level hardware models in VHDL and Verilog

The models have additional inputs and outputs over and above that of the algorithmic models. They are inputs Clock, Reset_N and Load, and the output Done. When Load is at logic 1 it signifies input data is available on inputs A and B, and are loaded into separate registers whose output signals are called A_hold and B_hold. The extra output signal, Done, switches to a logic 1 to signify the greatest common divisor has been computed. It takes a number of clock cycles to compute the GCD and is dependent upon the values of A and B.

The models are broken down into three **process** (VHDL)/**always** (Verilog) statements.

First **process**/**always** *statement* LOAD_SWAP. Infers two registers which operate as follows:

1) When Reset_N is at a logic 0, A_hold and B_hold are set to zero.
2) When not 1) and Load is at logic 1, data on A and B is loaded into A_hold and B_hold.
3) When not 1) or 2) and A_hold is less than B_hold, values on A_hold and B_hold are swapped, that is, A_hold and B_hold are loaded into B_hold and A_hold respectively.
4) When not 1), 2) or 3), A_hold is reloaded, that is, it keeps the same value. The value of A_hold - B_hold, from the second **process**/**always** statement, is loaded into B_hold.

Second **process**/**always** *statement* SUBTRACT. Tests to see if A_hold is greater than or equal to B_hold. If it is, the subtraction, A_hold - B_hold, occurs and the result assigned to A_New ready to be loaded into B_hold on the next rising edge of the clock signal. If A_hold is less than B_hold, then subtraction cannot occur and A_New is assigned the value B_hold so that a swap occurs after the next rising edge of the clock signal.

Third **process**/**always** *statement* WRITE_OUTPUT. Checks to see if the value of B_hold has reached zero. If it has, signal Done is set to logic 1 and the value of A_hold is passed to the output Y through an inferred multiplexer function.

It is a requirement of the problem to synthesize the generic model with 8-bit bus signals. This is easily achieved in Verilog model by setting the default parameter value Width to 8. This means it does not need to be separately instantiated before it can be synthesized and have the correct bit width. This is not the case in VHDL, which uses a generic. The value of the generic is only specified when the model is instantiated. Although the VHDL model will be instantiated in the test harness, the test harness is not synthesized. Therefore, in order to synthesize an 8-bit GCD circuit a separate synthesizable model must be used to instantiate the RTL level model which specifies the generic, Width, to be 8. The simulation test harness does not need to use this extra model as it too, will specify the generic, Width, to be 8.

VHDL - RTL

```
library IEEE;
use IEEE.STD_Logic_1164.all, IEEE.Numeric_STD.all;

entity GCD is
 generic (Width: natural);
 port ( Clock,Reset,Load: in  std_logic;
        A,B:      in  unsigned(Width-1 downto 0);
        Done:     out std_logic;
        Y:        out unsigned(Width-1 downto 0));
end entity GCD;

architecture RTL of GCD is
   signal A_New,A_Hold,B_Hold: unsigned(Width-1 downto 0);
   signal A_lessthan_B: std_logic;
begin

   ---------------------------------------------------------------------------
   -- Load 2 input registers and ensure B_Hold < A_Hold
   ---------------------------------------------------------------------------
   LOAD_SWAP: process (Clock)
   begin
      if rising_edge(Clock) then
         if (Reset = '0') then
            A_Hold <= (others => '0');
            B_Hold <= (others => '0');
         elsif (Load = '1') then
            A_Hold <= A;
            B_Hold <= B;
         elsif (A_lessthan_B = '1') then
            A_Hold <= B_Hold;
            B_Hold <= A_New;
         else
            A_Hold <= A_New;
         end if;
      end if;
   end process LOAD_SWAP;

   ---------------------------------------------------------------------------
   -- Subtract B_Hold from A_Hold if A_Hold >= B_Hold
   ---------------------------------------------------------------------------
   SUBTRACT: process (A_Hold, B_Hold)
   begin
      if (A_Hold >= B_Hold) then
         A_lessthan_B  <= '0';
         A_New <= A_Hold - B_Hold;
      else
         A_lessthan_B  <= '1';
         A_New <= A_Hold;
      end if;
   end process SUBTRACT;

   ---------------------------------------------------------------------------
   -- Greatest common divisor found if B_Hold = 0
   ---------------------------------------------------------------------------
   WRITE_OUTPUT: process (A_Hold,B_Hold)
   begin
      if (B_Hold = (others => '0')) then
         Done <= '1';
         Y <= A_Hold;
      else
         Done <= '0';
         Y <= (others => '0');
      end if;
   end process WRITE_OUTPUT;

 end architecture RTL;
```

Verilog - RTL

```
module GCD (Clock,Reset,Load,A,B,Done,Y);
      parameter Width = 8;
      input  Clock,Reset,Load;
      input  [Width-1:0] A,B;
      output Done;
      output [Width-1:0] Y;

      reg  A_lessthan_B,Done;
      reg  [Width-1:0] A_New,A_Hold,B_Hold,Y;

   //---------------------------------------------------------------------------
   // Load 2 input registers and ensure B_Hold < A_Hold
   //---------------------------------------------------------------------------
   always @(posedge Clock)
      begin: LOAD_SWAP
         if (Reset)
            begin
               A_Hold = 0;
               B_Hold = 0;
            end
         else if (Load)
            begin
               A_Hold = A;
               B_Hold = B;
            end
         else if (A_lessthan_B)
            begin
               A_Hold = B_Hold;
               B_Hold = A_New;
            end
         else
            A_Hold = A_New;
      end
   //---------------------------------------------------------------------------
   // Subtract B_Hold from A_Hold if A_Hold >= B_Hold
   //---------------------------------------------------------------------------
   always @(A_Hold or B_Hold)
      begin: SUBTRACT
         if (A_Hold >= B_Hold)
            begin
               A_lessthan_B = 0;
               A_New = A_Hold - B_Hold;
            end
         else
            begin
               A_lessthan_B = 1;
               A_New = A_Hold;
            end
      end
   //---------------------------------------------------------------------
   // Greatest common divisor found if B_Hold = 0
   //---------------------------------------------------------------------
   always @(A_Hold or B_Hold)
      begin: WRITE_OUTPUT
         if (B_Hold == 0)
            begin
               Done = 1;
               Y = A_Hold;
            end
         else
            begin
               Done = 0;
               Y = 0;
            end
      end
endmodule
```

5. Conclusions

The reasons for the importance of being able to model hardware in both VHDL and Verilog has been discussed. VHDL and Verilog has been extensively compared and contrasted in a neutral manner. A tutorial has been posed as a problem and solution to demonstrate some language differences and indicated that hardware modeled in one language can also be modeled in the other. Room did not allow test harness models to be included in this tutorial paper, but is shown in the book “HDL Chip Design” [1]. 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.

REFERENCES:

[1] HDL Chip Design, A Practical Guide for Designing, Synthesizing and Simulating ASICs and FPGAs using VHDL & Verilog by Douglas J Smith, published by Doone Publications.