Date:	**05/6/2020**	**Name:**	**SAFIYA BANU**
Course:	**DIGITAL DESIGN USING HDL**	**USN:**	**4AL16EC061**
Topic:	**Content** Verilog Tutorials and practice programs Building/ Demo projects using FPGA	**Semester & Section:**	**8th , B**
Github Repository:	**Safiya-Courses**		

Verilog is a HARDWARE DESCRIPTION LANGUAGE (HDL). A hardware description Language is a language used to describe a digital system, for example, a network switch, a microprocessor or a memory or a simple flip−flop. This just means that, by using a HDL one can describe any hardware (digital) at any level.

- Verilog supports a design at many different levels of abstraction. Three of them are very important:
  - ✓ Behavioral level
  - ✓ Register−Transfer Level
  - ✓ Gate Level

➢ Various stages of ASIC/FPGA

- Specification: Word processor like Word, Kwriter, AbiWord, Open Office.
- High Level Design: Word processor like Word, Kwriter, AbiWord, for drawing waveform use tools like waveformer or testbencher or Word, Open Office.

- Micro Design/Low level design: Word processor like Word, Kwriter, AbiWord, for drawing waveform use tools like waveformer or test bencher or Word. For FSM State CAD or some similar tool, Open Office.

RTL Coding : Vim, Emacs, conTEXT, HDL TurboWriter

1. Very fast on-chip (FPGA) demonstration

The top reason why I love FPGA design is that it is very fast to verify a design on FPGA. While ASICs could take several months only for tape-out and another latency for PCB design, everything is settled on FPGA and we just need to download the program file to FPGA using a specific FPGA programming tool and see how it works on FPGA. FPGA boards of Xilinx and Altera provide necessary IOs and additional support circuits such as LCD, single LEDs, 7-segment LEDs, communication ports (USB, UART, VGA, HDMI, PS2, FMC, etc.), ADCs, DACs, CODEC, etc. so that FPGA can easily communicate with other chips for the verification process.

2. Simple and fast design process on FPGA

Another great thing to say about FPGA is that the design process is pretty simple and really easy to learn. The design flow for ASICs is very complicated and time-consuming since it needs a lot of complex steps for designing, verification, and implementation. On the other hand, FPGA design process mostly avoids sophisticated and time-consuming steps like Floor-planing, Timing Analysis, Physical Implementation, etc. because FPGA is already a characterized and verified chip. Of course, when needed, FPGA vendors also provide necessary tools for floorplanning and timing analysis to enable users optimizing performance for niche very-demanding designs. In fact, FPGA design flow only takes several steps such as HDL design and coding, functional simulation, synthesis, timing or post-synthesis simulation if needed, and Place And Route. Furthermore, many FPGA design tools are free and very easy for users to learn and design. FPGA vendors provide free user guides and tutorials to facilitate user's learning process. It could take very short time for students to be familiar with FPGA design if they have a good background in digital logic design.

3. FPGA's programmability

The highlight feature of FPGA we obviously could not omit is its programmability. While ASICs or microcontrollers are fixed in term of hardware (it can be programmable at the software level), FPGAs can be programmable at the hardware level. We can program FPGAs to perform almost any digital complex functionality and reconfigure it to whatever we want in the future if needed. FPGAs can be programmed as a microprocessor, a microcontroller, DSPs, VGA controllers, digital filters, etc.

4. FPGA's high performance

Another superb feature in FPGA design is high performance. While processor-based ASICs or DSPs are sequential executed, FPGAs exploits the hardware parallelism to obtain a breakthrough performance for demanding designs. Thus, FPGAs provides faster implementations that processor-based ASICs could not match. The FPGA's parallelism can be effectively exploited to implement digital signal processing algorithms in order to speed up the processing time.

5. FPGA's flexibility

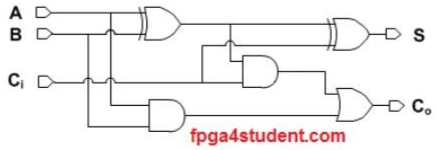
FPGAs are more and more flexible for designers to make their own applications. As mentioned above, FPGA vendors provide their own soft processors such as Xilinx's MicroBlaze, Altera's Nios II, etc. so that designers can be more flexible in design and programming process. Indeed, on FPGA, you can use the soft processors with the software-level-reconfigurable capability(C, C++, etc.) for low and average speed applications, and FPGA hardware accelerators with the hardware-level-reconfigurable capability(Verilog/ VHDL) for high-speed operations. Thus, designers can obtain a suitable embedded system which meets their design requirements. For example, when designing an embedded real-time tracking system, designers can use the soft processor for camera interface and FPGA hardware accelerators for tracking processing.

Verilog vs VHDL: Explain by Examples
What is FPGA Programming? FPGA vs Software programming
Recommended and affordable Xilinx FPGA boards for students
Recommended and affordable Altera FPGA boards for students
Recommended FPGA projects for students:
1. **What is FPGA? How does FPGA work?**
2. **Basys 3 FPGA OV7670 Camera**
3. **How to load text file or image into FPGA**

What is FPGA? FPGA stands for Field Programmable Gate Array. Let's analyze the term:
1. Field-Programmable: An FPGA is manufactured to be easily reconfigured by developers, designers or customers. To program an FPGA as a specific configuration, Verilog HDL or VHDL (Hardware Description Language) is used as the standard language for FPGA programming.
2. Gate-Array: An FPGA consists of an array of programmable logic gates/ blocks such as AND, OR, XOR, NOT, memory elements, DSP components, etc., and reconfigurable interconnects which are to connect logic gates together for performing a specific function.

Thus, FPGAs are nothing, but logic blocks and interconnects that can be programmable by Hardware Description Languages (Verilog HDL/ VHDL) to perform different complex functions.
In fact, FPGAs can be used to implement almost any DSP algorithm. Some FPGAs also obtain embedded soft-core processors such as Xilinx's MicroBlaze, Altera's Nios II, etc. so that we can use C, C++, etc. to program the processor like what we do with a microcontroller. Besides, the soft processors can communicate with hardware accelerators to speed up complex DSP operations so that we can obtain a better flexible embedded system for niche applications.

Let's take a very basic example on how to use an FPGA. Let's assume that you are designing a 1-bit full adder and you already obtained the logic diagram of the adder as shown in the figure below.



As mentioned above, there are necessary logic gates on FPGA such as XOR, AND and OR in order to implement the above adder on FPGA. To demonstrate the operation of the adder on FPGA, either Verilog or VHDL can be easily used to connect those gates together as shown in the logic diagram of the adder.

- **An example Verilog code for the adder:**

```
-- FPGA Project: What is an FPGA?
-- Verilog example code for Adder on FPGA
module fpga4student_adder(input A,B,Ci, output S,Co);
 wire tmp1,tmp2,tmp3;  //FPGA projects
 xor u1(tmp1,A,B);  // Verilog projects
 and u2(tmp2,A,B);
 and u3(tmp3,tmp1,Ci);
 or u4(Co,tmp2,tmp3);
 xor u5(S,tmp1,Ci);
 endmodule
```

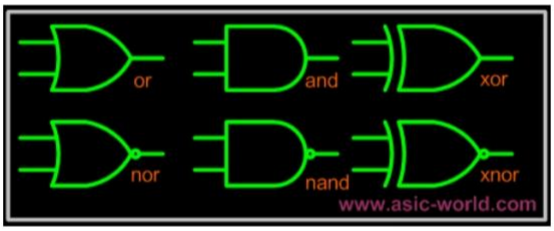
- **An example VHDL code for the adder:**

```
-- FPGA projects: What is an FPGA?
-- VHDL example code for adder on FPGA
 library ieee;
 use ieee.std_logic_1164.all;
 entity fpga4student_Adder is
   port( A, B, Ci : in std_logic;
      S, Co : out std_logic);
 end fpga4student_Adder;
 architecture structural of fpga4student_Adder is
 signal tmp1, tmp2, tmp3: std_logic;
 begin
   tmp1 <= A xor B;
   tmp2 <= A and B;
```

Introduction

Verilog has built in primitives like gates, transmission gates, and switches. This are rarely used for in design work, but are used in post synthesis world for modeling the ASIC/FPGA cells, this cells are then used for gate level simulation or what is called as SDF simulation. Also the output netlist formate from the synthesis tool which is imported into place and route tool is also in Verilog gate level primitives.

Gate Primitives



The gates have one scalar output and multiple scalar inputs. The 1st terminal in the list of gate terminals is an output and the other terminals are inputs.

Gate	Description
and	N–input AND gate
nand	N–input NAND gate
or	N–input OR gate
nor	N–input NOR gate
xor	N–input XOR gate
xnor	N–input XNOR gate

Examples

```
1 module gates();
2
3 wire out0;
4 wire out1;
5 wire out2;
6 reg in1,in2,in3,in4;
7
8 not U1(out0,in1);
9 and U2(out1,in1,in2,in3,in4);
10 xor U3(out2,in1,in2,in3);
11
12 initial begin
```



```
13   $monitor( "in1 = %b in2 = %b in3 = %b in4 = %b out0 = %b out1 = %b out2 = %b"
     ,in1,in2,in3,in4,out0,out1,out2);
14   in1 = 0;
15   in2 = 0;
16   in3 = 0;
17   in4 = 0;
18   #1 in1 = 1;
19   #1 in2 = 1;
20   #1 in3 = 1;
21   #1 in4 = 1;
22   #1 $finish;
23 end
```

Integer	Stored as
6'hCA	001010
6'hA	001010
16'bZ	ZZZZZZZZZZZZZZZZ
8'bx	xxxxxxxx

Real Numbers

- Verilog supports real constants and variables
- Verilog converts real numbers to integers by rounding
- Real Numbers can not contain 'Z' and 'X'
- Real numbers may be specified in either decimal or scientific notation
- < value >.< value >
- < mantissa >E< exponent >
- Real numbers are rounded off to the nearest integer when assignin

Example of Real Numbers

Real Number	Decimal notation
1.2	1.2
0.6	0.6
3.5E6	3,500000.0

Signed and Unsigned Numbers

Verilog Supports both the type of numbers, but with certain restrictions. Like in C language we don't have int and unint types to say if a number is signed integer or unsigned integer.

Any number that does not have negative sign prefix is a positive number. Or indirect way would be "Unsigned"

Negative numbers can be specified by putting a minus sign before the size for a constant number, thus become signed numbers. Verilog internally represents negative numbers in 2's compliment format. An optional signed specifier can be added for signed arithmetic.

Examples

Number	Description
32'hDEAD_BEEF	Unsigned or signed positive Number
-14'h1234	Signed negative number

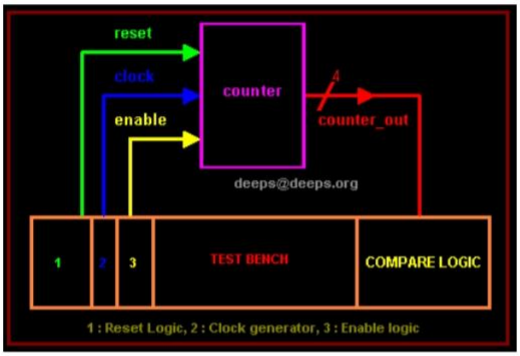
Below example file show how Verilog treats signed and unsigned numbers.

```
1  //////////////////////////////////////////////
2  // Signed Number Example
3  //
4  // Written by Deepak Kumar Tala
5  //////////////////////////////////////////////
6  module signed_number;
7
8  reg [31:0] a;
9
10 initial begin
11   a = 14'h1234;
12   $display ( "Current Value of a = %h" , a);
13   a = -14'h1234;
14   $display ( "Current Value of a = %h" , a);
15   a = 32'hDEAD_BEEF;
16   $display ( "Current Value of a = %h" , a);
17   a = -32'hDEAD_BEEF;
18   $display ( "Current Value of a = %h" , a);
```

```
47 endmodule // End of Module counter
```

Counter Test Bench

Any digital circuit, not matter how complex it is needs to be tested. For the counter logic, we need to provide clock, reset logic. Once counter is out of reset we toggle the enable input to counter, and check with waveform to see if counter is counting correctly. We do the same in Verilog.



Counter testbench consists of clock generator, reset control, enable control and compare logic. Below is the simple code of testbench without the compare logic.

```
 1 include "first_counter.v"
 2 module first_counter_tb();
 3 // Declare inputs as regs and outputs as wires
 4 reg clock, reset, enable;
 5 wire [3:0] counter_out;
 6
 7 // Initialize all variables
 8 initial begin
 9    $display ( "time\t clk reset enable counter" );
10    $monitor ( "%g\t %b %b %b %b" ,
11    $time, clock, reset, enable, counter_out);
12    clock = 1; // initial value of clock
```



```
13    reset = 0; // initial value of reset
14    enable = 0; // initial value of enable
15    #5 reset = 1; // Assert the reset
16    #10 reset = 0; // De-assert the reset
17    #5 enable = 1; // Assert enable
18    #100 enable = 0; // De-assert enable
19    #10 $finish; // Terminate simulation
20 end
21
22 // Clock generator
23 always begin
24    #5 clock = ~clock; // Toggle clock every 5 ticks
25 end
26
27 // Connect DUT to test bench
28 first_counter U_counter (
29 clock,
30 reset,
31 enable,
32 counter_out
33 );
34
35 endmodule
```

```
time    clk reset enable counter
0       1   0     0      xxxx
5       0   1     0      xxxx
10      1   1     0      xxxx
11      1   1     0      0000
15      0   0     0      0000
20      1   0     1      0000
21      1   0     1      0001
25      0   0     1      0001
30      1   0     1      0001
31      1   0     1      0010
35      0   0     1      0010
40      1   0     1      0010
41      1   0     1      0011
45      0   0     1      0011
```

Well I am also going to show how to write a **"hello world"** program in Verilog, followed by **"counter"** design in Verilog.

Hello World Program

```
1 //------------------------------------------------------
2 // This is my first Verilog Program
3 // Design Name : hello_world
4 // File Name : hello_world.v
5 // Function : This program will print 'hello world'
6 // Coder : Deepak
7 //------------------------------------------------------
8 module hello_world ;
9
10 initial begin
11    $display ( "Hello World by Deepak" );
12    #10 $finish;
13 end
14
15 endmodule // End of Module hello_world
```

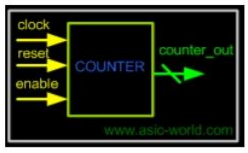
Words in green are comments, blue are reserved words, Any program in Verilog starts with reserved word module , In the above example line 7 contains module hello_world. (Note: We can have compiler pre-processor statements like `include, `define statements before module declaration)

Line 9 contains the initial block, this block gets executed only once after the simulation starts and at time=0 (0ns). This block contains two statements, which are enclosed within begin at line 7 and end at line 12. In Verilog if you have multiple lines within a block, you need to use begin and end.

Hello World Program Output

```
Hello World by Deepak
```

Counter Design Block



Counter Design Specs

- 4-bit synchronous up counter.
- active high, synchronous reset.
- Active high enable.

Counter Design

```
1 //------------------------------------------------------
2 // This is my second Verilog Design
3 // Design Name : first_counter
4 // File Name : first_counter.v
5 // Function : This is a 4 bit up-counter with
6 // Synchronous active high reset and
7 // with active high enable signal
8 //------------------------------------------------------
9 module first_counter (
10 clock , // Clock input ot the design
11 reset , // active high, synchronous Reset input
12 enable , // Active high enabel signal for counter
13 counter_out // 4 bit vector output of the counter
```

Simulation

Simulation is the process of verifying the functional characteristics of models at any level of abstraction. We use simulators to simulate the the Hardware models. To test if the RTL code meets the functional requirements of the specification, see if all the RTL blocks are functionally correct. To achieve this we need to write testbench, which generates clk, reset and required test vectors. A sample testbench for a counter is as shown below. Normally we spend 60–70% of time in verification of design.

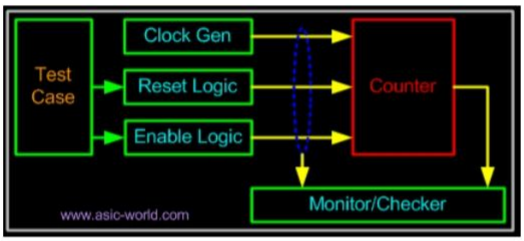


Figure : Sample Testbench Env

We use waveform output from the simulator to see if the DUT (Device Under Test) is functionally correct. Most of the simulators comes with waveform viewer, As design becomes complex, we write self checking testbench, where testbench applies the test vector, compares the output of DUT with expected value.

There is another kind of simulation, called **timing simulation**, which is done after synthesis or after P&R (Place and Route). Here we include the gate delays and wire delays and see if DUT works at rated clock speed. This is also called as **SDF simulation** or **gate level simulation**.

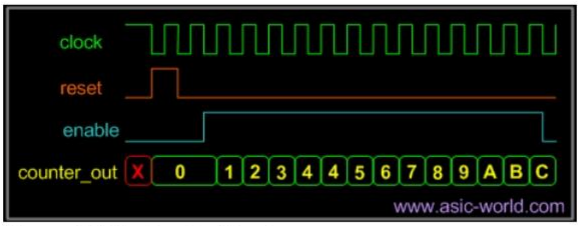
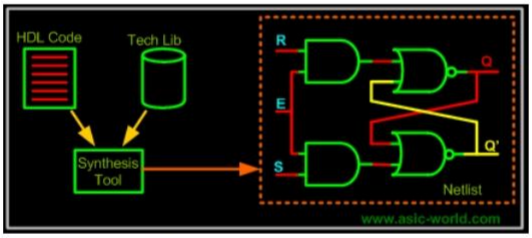


Figure : 4 bit Up Counter Waveform

Synthesis

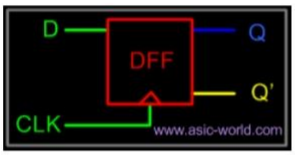
Synthesis is process in which synthesis tool like design compiler or Synplify takes the RTL in Verilog or VHDL, target technology, and constrains as input and maps the RTL to target technology primitives. Synthesis tool after mapping the RTL to gates, also do the minimal amount of timing analysis to see if the mapped design meeting the timing requirements. (Important thing to note is, synthesis tools are not aware of wire delays, they know only gate delays). After the synthesis there are couple of things that are normally done before passing the netlist to backend (Place and Route)

- **Formal Verification :** Check if the RTL to gate mapping is correct.
- **Scan insertion :** Insert the scan chain in the case of ASIC.



Introduction

Verilog is a **HARDWARE DESCRIPTION LANGUAGE (HDL)**. A hardware description Language is a language used to describe a digital system, for example, a network switch, a microprocessor or a memory or a simple flip–flop. This just means that, by using a HDL one can describe any hardware (digital) at any level.




```
1 // D flip–flop Code
2 module d_ff ( d, clk, q, q_bar);
3 input d ,clk;
4 output q, q_bar;
5 wire d ,clk;
6 reg q, q_bar;
7
8 always @ (posedge clk)
9 begin
10   q <= d;
11   q_bar <= !d;
12 end
13
14 endmodule
```

One can describe a simple Flip flop as that in above figure as well as one can describe a complicated designs having 1 million gates. Verilog is one of the HDL languages available in the industry for designing the Hardware. Verilog allows us to design a Digital design at Behavior Level, Register Transfer Level (RTL), Gate level and at switch level. Verilog allows hardware designers to express their designs with behavioral constructs, deterring the details of implementation to a later stage of design in the final design.

Many engineers who want to learn Verilog, most often ask this question, how much time it will take to learn Verilog?, Well my answer to them is **"It may not take more then one week, if you happen to know at least one programming language"**.

Design Styles

Verilog like any other hardware description language, permits the designers to design a design in either Bottom–up or Top–down methodology.

Bottom–Up Design

The traditional method of electronic design is bottom–up. Each design is performed at the gate–level using the standard gates (Refer to the Digital Section for more details) With increasing

complexity of new designs this approach is nearly impossible to maintain. New systems consist of ASIC or microprocessors with a complexity of thousands of transistors. These traditional bottom–up designs have to give way to new structural, hierarchical design methods. Without these new design practices it would be impossible to handle the new complexity.

Top–Down Design

The desired design–style of all designers is the top–down design. A real top–down design allows early testing, easy change of different technologies, a structured system design and offers many other advantages. But it is very difficult to follow a pure top–down design. Due to this fact most designs are mix of both the methods, implementing some key elements of both design styles.

Figure shows a Top–Down design approach.

Task :

Implement a verilog module to count number of 0's in a 16 bit number in compiler.

code

```
module num_zero(input [15:0]A, output reg [4:0]zeros);
  integer i;
  always@(A)
    begin
      zeros=0;
      for(i=0;i<16;i=i+1)
        zeros=zeros+A[i];
    end
endmodule
```

test bench code

```
module test;
  reg [15:0]A;
  wire [4:0] zeros;
num_zero out (.A(A), .zeros(zeros));
  initial begin
    $dumpfile("dumo.vcd");
    $dumpvars(1,test);
    A=16'hFFFF; #100;
    A=16'hF56F; #100;
    A=16'h3FFF; #100;
    A=16'h0001; #100;
    A=16'hF10F; #100;
    A=16'hF822; #100;
```

```
    A=16'h7ABC; #100;
  end
endmodule
```