Lab 4

Introduction to Verilog HDL

Gate Level Modeling

Learning objectives of this lab,

* Gate Level Modeling
* Pre-Lab

1. Verilog
   1. About Verilog

Verilog HDL is a programming 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 (digital) hardware at any level.

* 1. Design Methodologies

There are two basic types of digital design methodologies: a top-down design methodology and a bottom-up design methodology

* + 1. Top-Down Design Methodology

In a top-down design methodology, we define the top-level block and identify the sub-blocks necessary to build the top-level block. We further subdivide the sub-blocks until we come to leaf cells, which are the cells that cannot further be divided

* + 1. Bottom-up Design Methodology

In a bottom-up design methodology, we first identify the building blocks that are available to us. We build bigger cells, using these building blocks. These cells are then used for higher-level blocks until we build the top-level block in the design. In a bottom-up design methodology, we first identify the building blocks that are available to us. We build bigger cells, using these building blocks. These cells are then used for higher-level blocks until we build the top-level block in the design

* 1. Abstraction Level

Verilog descriptions can span multiple levels of abstraction i.e. levels of detail, to describe a digital system and can be used for different purposes at various stages in the design process.

Following are different abstraction levels:

* Switch Level (Transistor Level)
* Gate Level (AND, OR gates etc.)
* Boolean Equation Level (Continuous Assignment)
* Behavioral Level
  + - Procedural Assignment or Blocking Assignment (Combinational Circuits)
    - Register Transfer Level or Non-Blocking Assignment (Sequential Circuits)
    - Algorithmic (Combination & Sequential). Not Necessarily Synthesizable

In this lab we will practice Gate Level and Continuous Assignment. Both have one thing in common, i.e. the detail of the circuit should already be known before using these two abstraction levels. In the next lab we will use an abstraction level where only the relationship between input and output is described, and the circuit is synthesized by the synthesis tool (software).

* 1. Writing Code
     1. Code Structure

Verilog code is a written as a module and its test bench. Module represent block of digital circuit performing the desired functionality. It has input and output ports, written along with its declaration. A test bench is used to test the module by providing an input and checking the corresponding output.

* + 1. Module

A module is the basic building block in Verilog. A module can be an element or a collection of lower-level design blocks. Typically, elements are grouped into modules to provide common functionality that is used at many places in the design. A module provides the necessary functionality to the higher-level block through its port interface (inputs and outputs), but hides the internal implementation.

module my\_design (module\_ports);

// Declarations of ports go here

// Functional details go here

endmodule

The mode of a module port (input or output) is not determined by the order in which it appears in the list (as opposed to primitives). Its mode is defined inside the module

module my\_half\_adder(sum, c\_out, a, b);

output sum, c\_out;

input a,b;

xor M1(sum, a, b); //Instantiating primitive XOR

and M2(c\_out, a, b);//Instantiating primitive AND

//Any instance name can be //given. M1 and M2 are not //keywords. You may use your //name

endmodule

* + 1. Testbench

In order to test the module written, there should be a mechanism to supply input to the input ports and then check the result at the output ports of module. A testbench, which is also a code (module) written in verilog, provides this functionality.

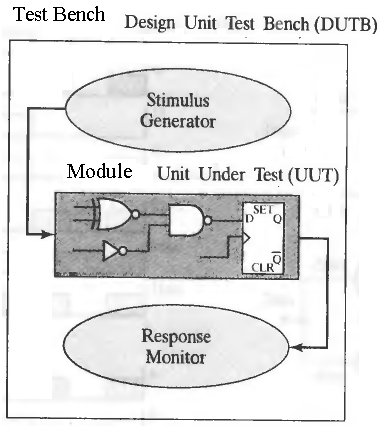


Figure 4.1

A testbench module doesn’t have any inputs and outputs. The desired module to be tested is instantiated inside it, just as the primitives(XOR & AND gates) were instantiated in our previous code. Before instantiating the module variables need to be defined. Variables which have the task of acting as a stimulus are declared as type reg, while the ones that are used to check the response are declared as type wire.

In order to test the module the stimulus (reg variables) should be given different values during testing period. This is accomplished by a single-pass Verilog behavior, declared by the keyword initial and accompanied by statements. It is enclosed by the block statement keyword pair begin and end if there are more than 1 instructions in the behavior.

module test\_my\_half\_adder;

wire sum, c\_out; //Outputs of UUT (in this case

// ‘module my\_half\_adder’) declared as //wire

reg a, b; // Inputs of UUT declared as ‘reg‘

my\_half\_adder AA(sum, c\_out, a, b); ­//Instantiating

­ //UUT

initial //Simulation will end

#100 $finish; // at 100ns

initial begin //Initially a=x, b=x

#10 a = 0; b = 0; //After 10ns a=0, b=0

#l0 b = 1; //After 20ns a=0, b=1

#10 a = 1; //After 30ns a=1, b=1

#l0 b = 0; //After 40ns a=1, b=0

end

endmodule

A testbench can have multiple initial behaviors which run in parallel. For example the testbench given above can be modified as

module test\_my\_half\_adder;

wire sum, c\_out; //Outputs of UUT (in this case

// ‘module my\_half\_adder’) declared as //wire

reg a, b; // Inputs of UUT declared as ‘reg‘

my\_half\_adder AA(sum, c\_out, a, b); ­//Instantiating

­ //UUT

initial begin //Simulation will end

#100 $finish; // at 100ns

end

initial begin //Initially a=x, b=x

#10 a = 0; //After 10ns a=0

#20 a = 1; //After 30ns a=1

end

initial begin //Initially a=x, b=x

#10 b = 0; //After 10ns b=0

#l0 b = 1; //After 20ns b=1

#20 b = 0; //After 40ns a=1, b=0

end

endmodule

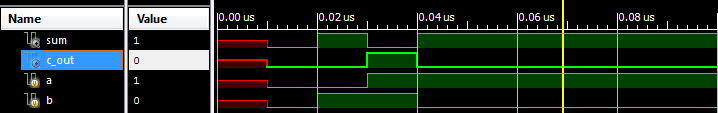


Fig 4.2 (Testbench output)

* + 1. Ports and Nets

The input and output ports are declared by the keywords input and output. A vector (more than 1 bit) is declared by enclosing the size in square brackets. For example a 4-bit input ‘a’ is declared by input [3:0] a;. The individual bits in a vector can be excessed. For example a[2:1] will select 2nd and 3rd bit from the 4-bit vector a[3:0].

For a multilayer circuit the intermediate connecting wires (nets) are declared by the keyword wire.

Verilog uses a four-valued logic system having the symbols: 0, 1, x, and z. The value x represents a condition of ambiguity in which the simulator cannot determine whether the value of the signal is 0 or 1. This happens, for example, when a net is driven by two primitives that have opposing output values. The logic value z denotes a condition in which a wire is disconnected from its driver.

* + 1. Primitives

Verilog has built in primitives like gates which are fundamental in gate level simulation. These have n-inputs and single output.

|  |  |
| --- | --- |
| AND Gate | and(y,x1,x2,……,xn) |
| OR Gate | or(y,x1,x2,……,xn) |
| NAND Gate | nand(y,x1,x2,……,xn) |
| NOR Gate | nor(y,x1,x2,……,xn) |
| XORGate | xor(y,x1,x2,……,xn) |
| XNOR Gate | xnor(y,x1,x2,……,xn) |

NOT gate has just one input ; not(y,x). Note that in primitives output is always written first in the arguments. However in a user defined module, any order is allowed.

* + 1. Instantiating user-defined Module

A module can instantiate another smaller module in it. This approach is convenient when coding large systems. For example, a Full Adder module (which has three inputs) can be written by just connecting two Half Adder Modules.

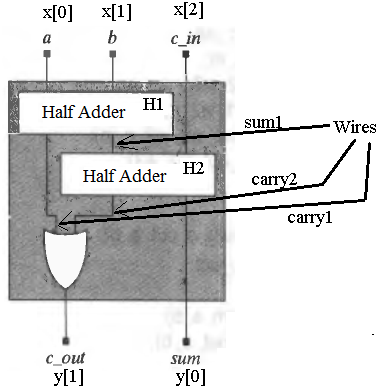


Fig 4.3

module my\_full\_adder(input [2:0] x, output [1:0] y);

//Different style + use of vectors

// x is 3-bit input & y is 2-bit output ,sum and carry;

wire sum1,carry1,carry2;

my\_half\_adder H1(sum1, carry1, x[0], x[1]);

my\_half\_adder H2(y[0], carry2, sum1, x[3]);

or (y[1],carry1,carry2);

endmodule

Write the testbench yourself.

* In-Lab

**Tasks(Use Xilinx)**

* **Write the gate level description of NAND SR-Latch.**

****

* **Write the gate level description of D-latch using SR-Latch module**

****

* Post-Lab
* **Write the Verilog HDL description of D-Type Flip-Flop using D-latches**

****

* **Your lab report, a .doc file, should contain properly commented Post-Lab task code, with Screenshots(of print preview) of Schematic and waveforms, and Critical Analysis.**
* **The report must have a title page.**
* **Name the .doc file RegNo.docx; eg SP14-BCE-99.docx**
* **Upload it on portal**