INTRODUCTION TO VERILOG

EC39004 : VLSI LABORATORY

January 7, 2019

Introduction

- **CAD tools are based on Hardware Description Language (HDL)**
- **A CAD tool transforms its HDL input into a HDL output that contains more detailed information about the hardware**
- **Typical transformation steps are:**

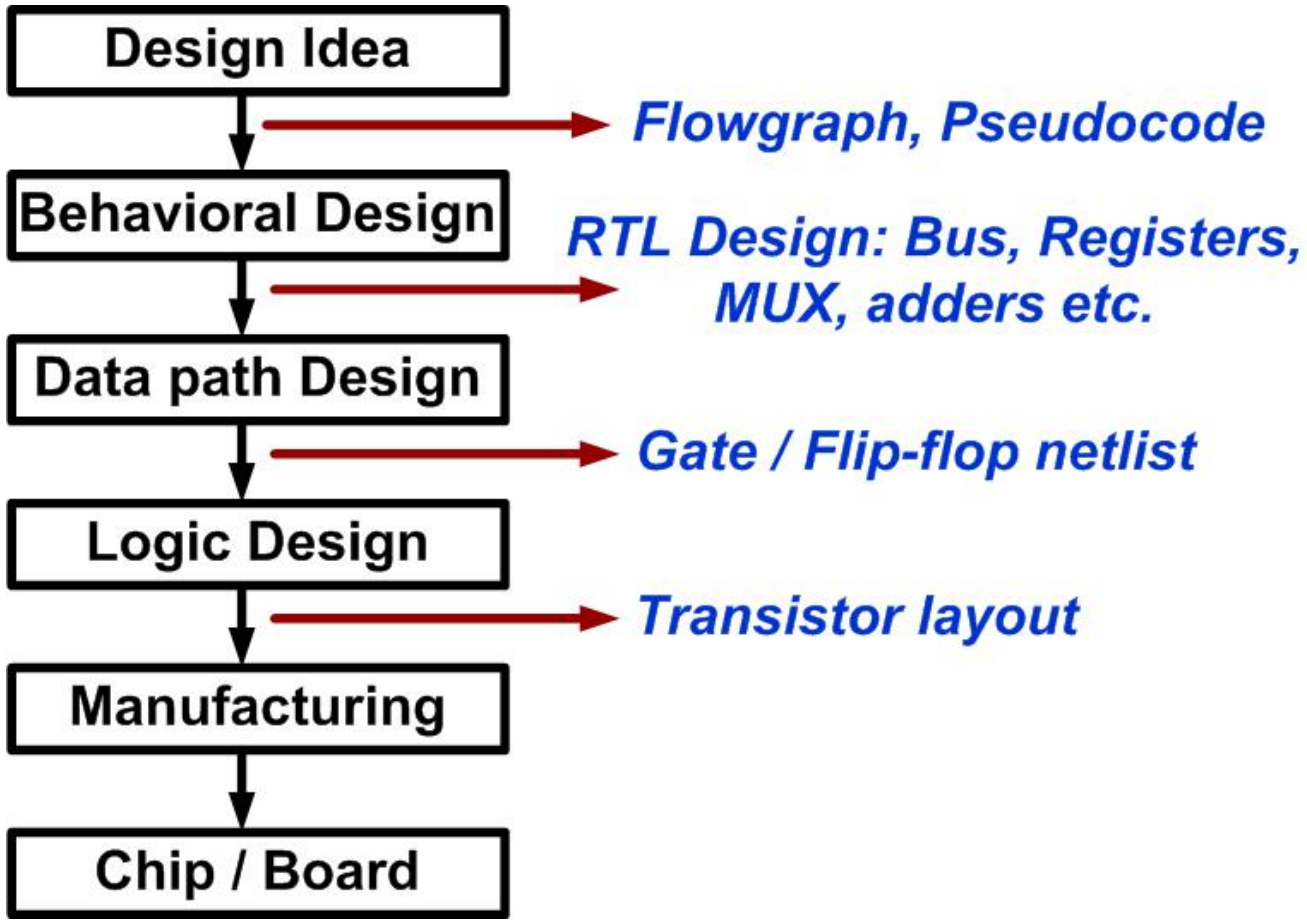
i) Behavioral level to Register Transfer Level (RTL)

ii) RTL to gate level

iii) Gate level to transistor level

iv) Transistor level to layout

Simplistic View of Design Flow



Hardware Description Language (HDL)

- **Two competing HDLs: Verilog and VHDL (Very high speed integrated circuit HDL)**
- **Designs are created typically using HDLs, which get transformed from one level of abstraction to the next as the design flow progresses**
- **There are other HDLs like SystemC, System Verilog etc.**
- **HDL can describe a digital system as a set of modules**
- **Each of the modules will have an interface to other modules, in addition to its description**

HDL (contd…)

- **There are two ways to specify a module:**

i) By specifying its internal logic structure (called structural representation)

ii) By describing its behavior in a program-like manner (called behavioral representation) The modules are interconnected using nets which allow them to work with each other

- **After specifying the system in Verilog, we can**

  i) Simulate the system and verify the operation

  ii) Use a synthesis tool to map it to hardware (typically Application Specific Integrated Circuit (ASIC) or Field Programmable Gate Array (FPGA))

IMPORTANCE OF VERILOG

- HDLs have many advantages compared to traditional schematic-based design.
- Designs can be described at a very abstract level by use of HDLs. Designers can write their RTL description without choosing a specific fabrication technology.
- By describing designs in HDLs, functional verification of the design can be done early in the design cycle.
- Designing with HDLs is analogous to computer programming. A textual description with comments is an easier way to develop and debug circuits.

IMPORTANCE OF VERILOG(contd..)

- Verilog HDL is a general-purpose hardware description language that is easy to learn and easy to use. It is similar in syntax to the C programming language
- Verilog HDL allows different levels of abstraction to be mixed in the same model. Thus, a designer can define a hardware model in terms of switches, gates, RTL, or behavioral code.
- Most popular logic synthesis tools support Verilog HDL. This makes it the language of choice for designers.
- designing a chip in Verilog HDL allows the widest choice of vendors.

Concept of Verilog module

- In Verilog, the basic unit of hardware is called a "*module*"
- Modules cannot contain *definitions* of other modules
- A module can however be *instantiated* within another module
- Creation of *hierarchy* of Verilog modules in a total description of the entire hardware is permissible

Basic syntax of “module” definition

```
module module_name (list_of_ports);
        input <variable_list>;        / /   input declarations  output
        <variable_list>;                    / /  output declarations  wire
        <variable_list>; / /    Internal net variables
         reg <variable_list>; / /    Register variables
        Parallel statements; / /  Hardware description
endmodule
```

A simple AND gate

```
module simpleand (f, x, y);
input x, y;
output f;
assign f =  x & y;
endmodule
```

OR

```
module simpleand (output f, input x,   input y);
assign f =  x & y;
endmodule
```

Comments

- **“assign f = x & y” typically represents an AND function which is the behavior of the AND gate: behavioral description**
- **It does not necessarily mean that a 2-input AND gate will be realized**
- **The synthesis tool will decide how to realize f :**

  a) Use a single 2-input AND gate

  b) Use a NAND gate followed by a NOT gate

  c) Use three NOR gates

  d) Use a multiplexer

Two level circuit

```
module two_level (a, b, c, d, f);
input a, b, c, d;
output f;
wire t1, t2;
assign t1 =  a & b;
assign t2 = ~ (c |  d);
assign f =  t1 ^ t2;
endmodule
```

Module instantiation

- A module provides a template from which you can create actual objects. When a module is invoked, Verilog creates a unique object from the template. Each object has its own name, variables, parameters and I/0 interface.
- Two types of instantiation:

1. **Implicit or Positional association**

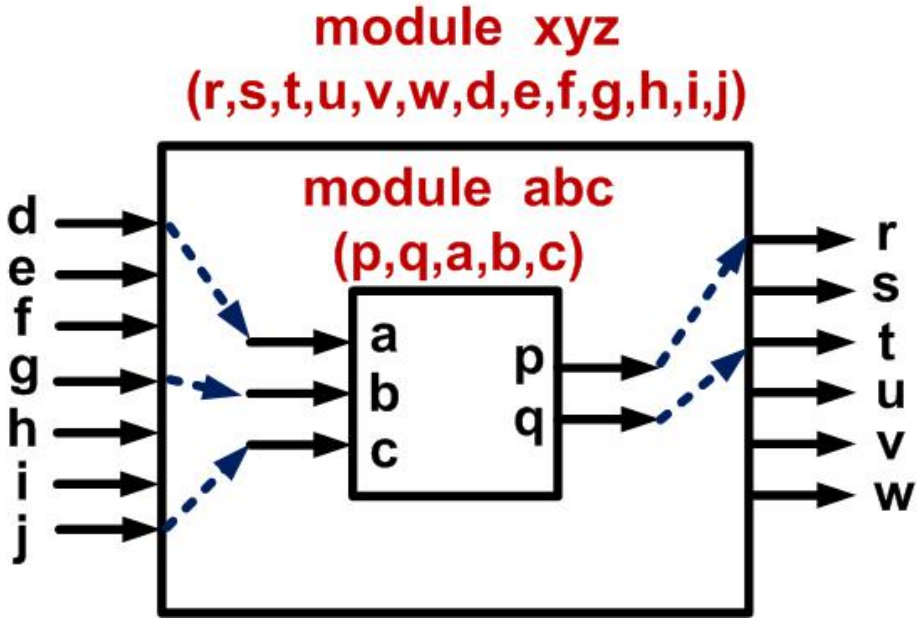
Example:

abc stage0 (r, t, d, g, j);

2. **Explicit or Named association**

Example:

abc stage0 (.a(d), .c(j), .b(g), .p(r), .q(t));



Data Types in Verilog

A variable in Verilog belongs to one of two data types:

a) NET (wire, tri, wand, wor, supply0, supply1)

- **Must be continuously driven**
- **Cannot be used to store a value**
- **Used to model connection between continuous assignments and instantiations**

b) REGISTER (reg, integer)

- **Retains the last value assigned to it**
- **Often used to represent storage elements, but sometimes it can translate to combinational circuits also**

Different styles of design representation

Behavioral

- **Specify the functionality of the design in terms of its behavior which may be specified by:**
  - **a) Boolean expression or truth table**
  - **b) Finite State Machine (FSM) modeling**

Structural

- **i) Specifies how components are interconnected**
- **ii) The description is a list of modules and their interconnection called a NETLIST.**
- **iii) Modules are specified typically at the gatelevel**

Different styles of design representation(contd..)

Physical level

- This is the lowest level of abstraction provided by Verilog. A module can be implemented in terms of switches, storage nodes, and the interconnections between them. Design at this level requires knowledge of switch-level implementation details.

```
module my-nor (out, a, b) ;
output out;
input a, b;
//internal wires
wire c;
//set up power and ground lines
supply1 pwr; //pwr is connected to Vdd (power supply)
supply0 gnd ; //gnd is connected to Vss(ground)
//instantiate pmos switches
pmos (c, pwr, b);
pmos (out, c, a);
//instantiate nmos switches
nmos (out, gnd, a);
nmos (out, gnd, b);
   endmodule
```

Example: Ripple Carry Adder

- **Design conception: Follow a top-down approach**

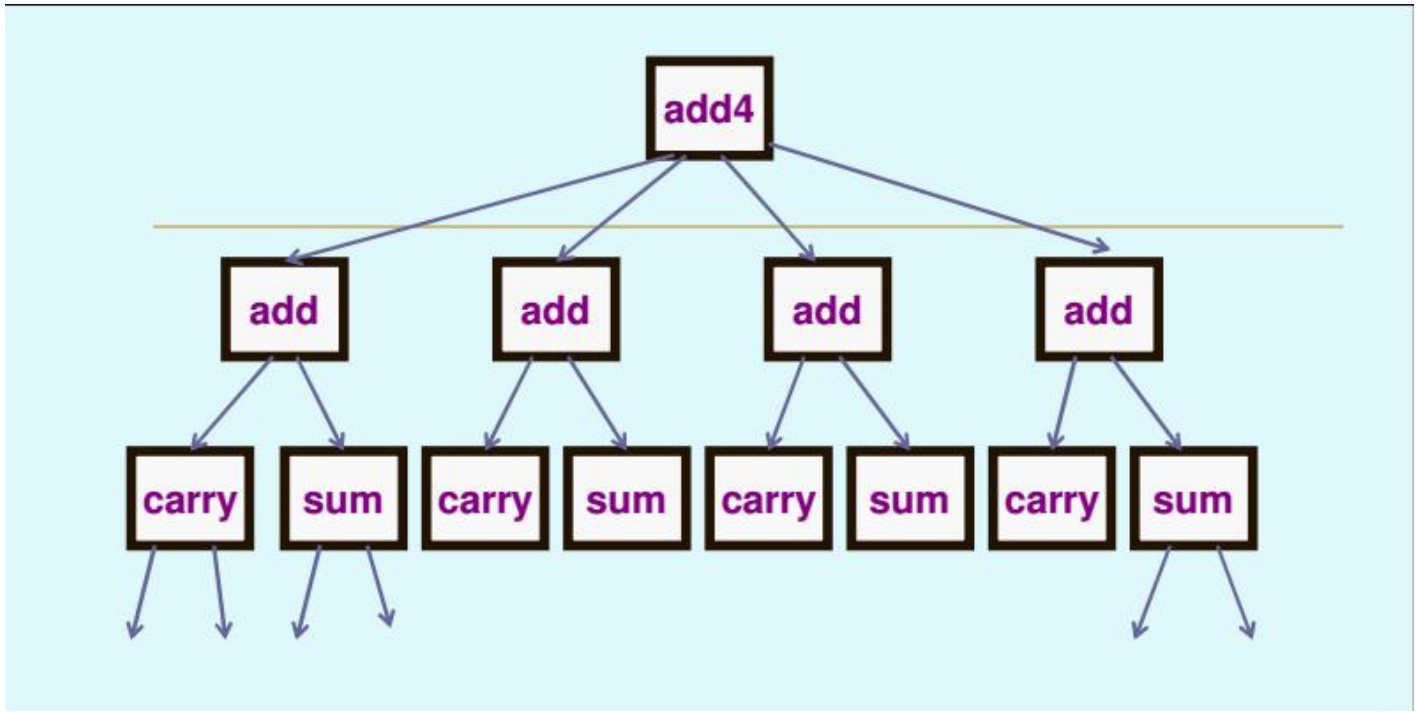


Figure: Top-down approach of Ripple Carry Adder.

Behavorial modelling

1. The “SUM” module

```
module sum (output s, input a, input b, input cy_in);
assign s= a^b^cy_in;
endmodule
```

2. The “CARRY” module

```
module carry (output co, input a, input b, input cy_in);
assign co= (a&b)|(b&cy_in)|(a&cy_in);
endmodule
```

Structural modelling

SUM MODULE

```
module sum(X, Y, Ci, S, Co);
input X, Y, Ci;
output S, Co;
wire w1,w2,w3;
xor G1(w1, X, Y);
xor G2(S, w1, Ci);
endmodule
```

CARRY MODULE

```
module  carry(X, Y, Ci, S, Co);
input X, Y, Ci;
output S, Co;
wire w1,w2,w3
and G3(w1, Y, Ci);
and G4(w2, X, Y);
and G4(w3, X, CI);
or G6(Co, w1, w2,w3);
endmodule
```

Structural modelling(contd..)

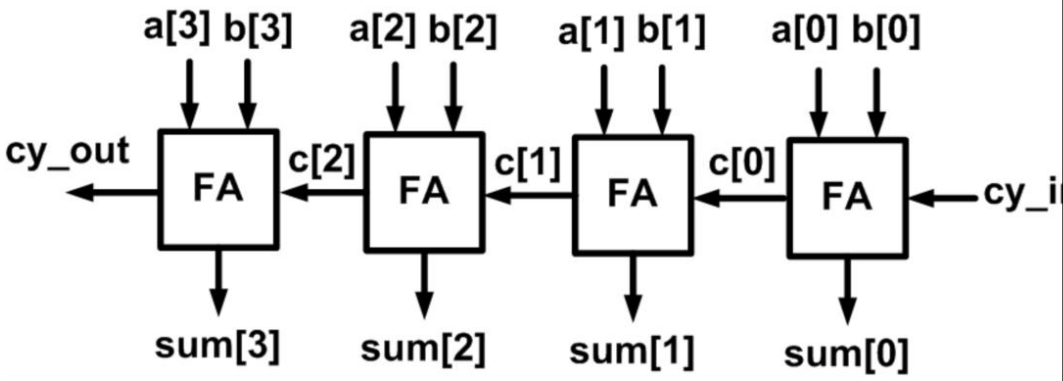
The 1-bit "FULLADDER" module

```
module fa (output s, output cout, input a, input b,  input cy_in);
sum  stage0 (.s(s), .a(a), .b(b), .cy_in(cy_in));
carry stage1 (.co(cout), .a(a), .b(b), .cy_in(cy_in));
endmodule
```

Structural modelling(contd..)

The 4-bit "RIPPLE CARRY ADDER"

```
module rca4 (output cy_out, output[3:0] sum,
input [3:0] a, input [3:0] b, input cy_in);
wire [2:0] c; / / Net declaration
fa stage0 (.s(sum[0]), .cout(c[0]),.a(a[0]),.b(b[0]), .cy_in(cy_in));
fa stage1 (.s(sum[1]), .cout(c[1]),.a(a[1]), .b(b[1]), .cy_in(c[0]));
fa stage2 (.s(sum[2]), .cout(c[2]),.a(a[2]),.b(b[2]), .cy_in(c[1]));
fa stage3 (.s(sum[3]), .cout(cy_out),.a(a[3]),.b(b[3]), .cy_in(c[2]));
endmodule
```



Architecture of Synthesized Ripple Carry Adder

Simulation Procedure

- **After specifying the system in Verilog, we can:**

a) SIMULATE the system and verify the operation

- Just like running a program written in some high-level language
- Requires a test benchor a test harness that specifies the inputs that are to be applied and the way the outputs are to be displayed

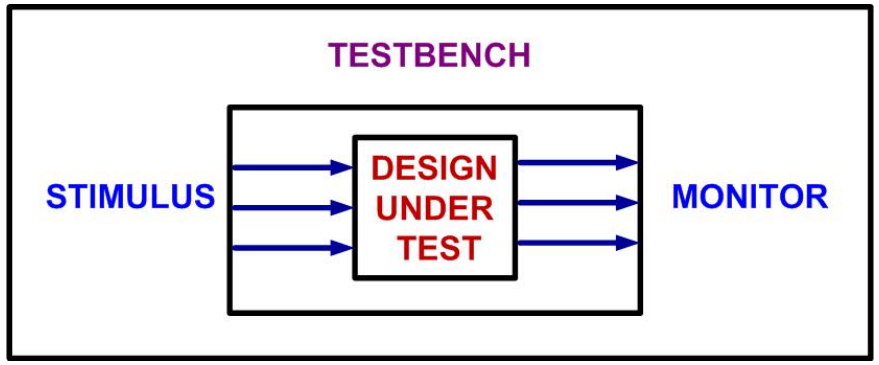
b) Once the output results are true, the design can be actually mapped to hardware: here signals can be actually applied from source (eg. signal generator), and response is evaluated by some equipment (eg. Oscilloscope or logic analyzer)

How to Simulate Verilog Modules?

Using a test bench to verify the functionality of a design coded in Verilog (called Design-Under-Test or DUT), comprising of:

- → **A set of stimulus for the DUT**
- → **A monitor which captures or analyzes the outputs of the DUT**

Both the inputs and the outputs of the DUT need to be connected to the test bench



Concept of Design Under Test.

How to Write Test bench?

- **Create a template**

Declare inputs to the DUT as “reg” and the o/ps as “wire”

Instantiate the DUT

Initialization and Monitoring

Assign some known values to the DUT inputs

Monitor the DUT outputs for functional verification

A Few Simulator Directives

$display (“<format string>”, expr1,expr2,…)

- **Used to print immediate values of text or variables to stdoutas soon as it is encountered**
- **Syntax is very similar to “printf” statement in C**
- **Additional format specifiers are supported like “b” (binary), “h” (hexadecimal) etc…**
- **Example: $display(“%b,%d,%h”, x,y,z);**

A Few Simulator Directives

$monitor("<format string>", expr1,expr2,…)

i)Similar in syntax to $display, but does not print immediately

ii)It will print the value(s) whenever the value of some variable(s) in the given list changes

iii)Has the functionality of event-driven point

Example: $monitor("%b,%d,%h", x,y,z);

$time : Returns the simulation time stamp

$finish : Terminates the simulation process

Verilog Code for Full Adder

Verilog Code for a full adder

```
module adder(  input a,b,c,  output s,cout);
assign s=a^b^c;
assign cout=(a&b)|(b&c)|(a&c);
endmodule
```

Test Bench

- **Testbench**

```
module tester;
reg a, b, c; / /   Inputs
wire s, cout; / /  Outputs
adder dut ( .a(a), .b(b), .c(c), .s(s), .cout(cout) );  / /  Instantiate the  Design Under Test (DUT)
initial begin
$monitor(" Time=%5d, a=%b, b=%b, c=%b, s=%b, cout=%b\n",
$time, a,b,c,s,cout);
 / /  Initialize Inputs
 #10;       a =  0; b =  0; c =  0; #100; a=0; b=0; c=1; #100; b=1; #100; a=1;
  #100;    b=0; #100; a=0; #100; c=0; #40
  $finish;
end  endmodule
```

Behavioral Simulation Output

- **Time=0, a=x, b=x, c=x, s=x, cout=x**
- **Time=10, a=0, b=0, c=0, s=0, cout=0**
- **Time=110, a=0, b=0, c=1, s=1, cout=0**
- **Time=210, a=0, b=1, c=1, s=0, cout=1**
- **Time=310, a=1, b=1, c=1, s=1, cout=1**
- **Time=410, a=1, b=0, c=1, s=0, cout=1**
- **Time=510, a=0, b=0, c=1, s=1, cout=0**
- **Time=610, a=0, b=0, c=0, s=0, cout=0**

Post Route Simulation o/p

Time=0, a=x, b=x, c=x, s=x, cout=x
Time=10, a=0, b=0, c=0, s=x, cout=x
Time=15, a=0, b=0, c=0, s=x, cout=0
Time=15, a=0, b=0, c=0, s=0, cout=0
Time=110, a=0, b=0, c=1, s=0, cout=0
Time=115, a=0, b=0, c=1, s=1, cout=0
Time=210, a=0, b=1, c=1, s=1, cout=0
Time=215, a=0, b=1, c=1, s=1, cout=1
Time=215, a=0, b=1, c=1, s=0, cout=1

Test Bench Waveform

- Test Bench Waveform of Full Adder

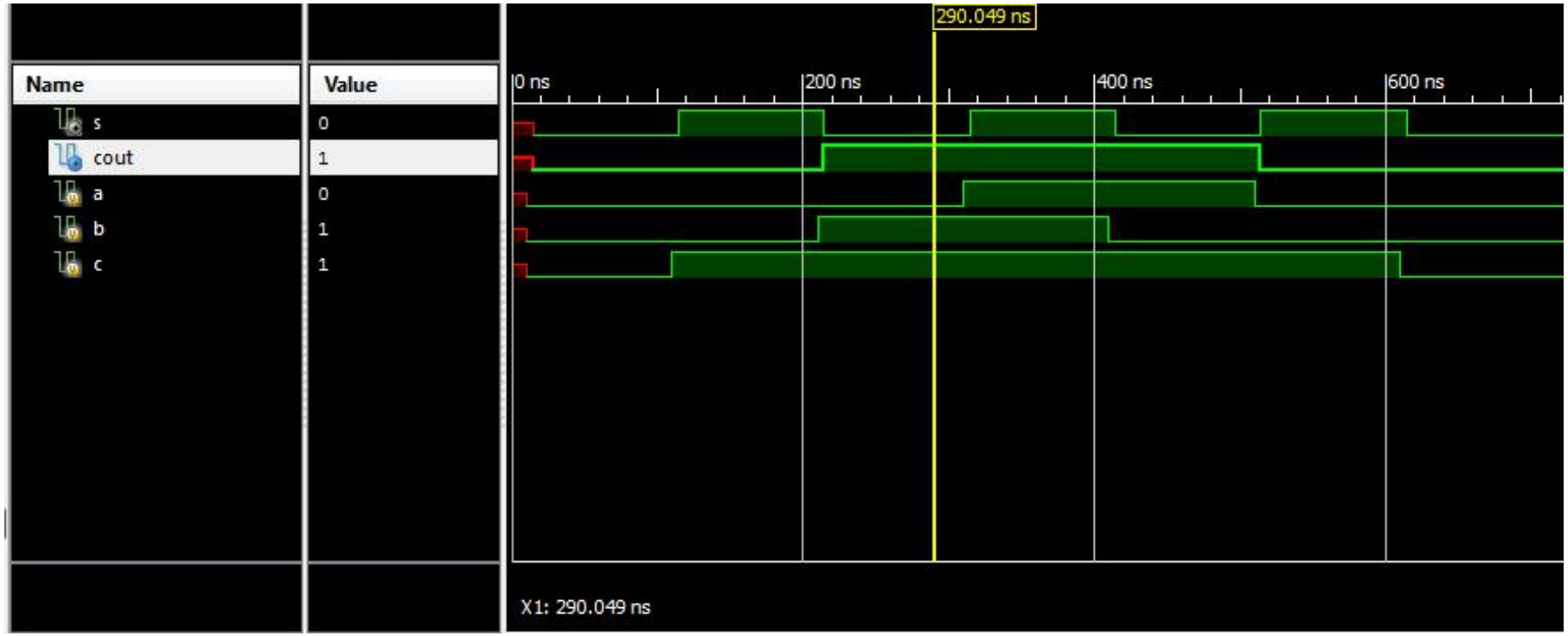


Figure: Test Bench Waveform of Full Adder.

Study Materials

- Verilog Hdl by Samir Palnitkar
- Fundamentals of Logic Design with Verilog by Stephen Brown and Zvonko Vranesic (Tata Mc Graw Hill Pub.)
- Advanced Digital Design with the Verilog HDL by Michael D. Ciletti
- Introduction to Logic Synthesis using Verilog HDL by Resse and Thronton (Morgan Claypool Pub.)
- FPGA Prototyping by Verilog Examples by Pong P. Chu