FREE eBook

LEARNING verilog

Free unaffiliated eBook created from
Stack Overflow contributors.

#verilog

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: verilog

It is an unofficial and free verilog ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official verilog.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with verilog

Remarks

Verilog is a hardware description language (HDL) that is used to design, simulate, and verify digital circuitry at a behavioral or register-transfer level. It is noteworthy for a reasons that distinguish it from "traditional" programming languages:

- There are two types of assignment, blocking and non-blocking, each with their own uses and semantics.
- Variables must be declared as either single-bit wide or with an explicit width.
- Designs are hierarchical, with the ability to instantiate modules that have a desired behaviour.
- In simulation (not typically in synthesis), `wire` variables may be in one of four states: 0, 1, floating (`z`), and undefined (`x`).

Versions

Version	Release Date
Verilog IEEE-1364-1995	1995-01-01
Verilog IEEE-1364-2001	2001-09-28
Verilog IEEE-1364.1-2002	2002-12-18
Verilog IEEE-1364-2005	2006-04-07
SystemVerilog IEEE-1800-2009	2009-12-11
SystemVerilog IEEE-1800-2012	2013-02-21

Examples

Installation or Setup

Detailed instructions on getting Verilog set up or installed is dependent on the tool you use since there are many Verilog tools.

Introduction

Verilog is a hardware description language (HDL) used to model electronic systems. It most commonly describes an electronic system at the register-transfer level (RTL) of abstraction. It is also used in the verification of analog circuits and mixed-signal circuits. Its structure and main

principles (as described below) are designed to describe and successfully implement an electronic system.

- Rigidity
  An electronic circuit is a physical entity having a fixed structure and Verilog is adapted for that. Modules (module), Ports(input/output/inout) , connections (wires), blocks (@always), registers (reg) are all fixed at compile time. The number of entities and interconnects do not change dynamically. There is always a "module" at the top-level representing the chip structure (for synthesis), and one at the system level for verification.
- Parallelism
  The inherent simultaneous operations in the physical chip is mimicked in the language by always (most commmon), initial and fork/join blocks.

```
 module top();
 reg r1,r2,r3,r4; // 1-bit registers
   initial
   begin
     r1 <= 0 ;
   end
   initial
   begin
     fork
        r2 <= 0 ;
        r3 <= 0 ;
     join
   end
    always @(r4)
       r4 <= 0 ;
endmodule
```

All of the above statements are executed in parallel within the same time unit.

- Timing and Synchronization
  Verilog supports various constructs to describe the temporal nature of circuits. Timings and delays in circuits can be implemented in Verilog, for example by #delay constructs. Similarly, Verilog also accommodates for synchronous and asynchronous circuits and components like flops, latches and combinatorial logic using various constructs, for example "always" blocks. A set of blocks can also be synchronized via a common clock signal or a block can be triggered based on specific set of inputs.

  ```
  #10   ;                 // delay for 10 time units
  always @(posedge clk )  // synchronous
  always @(sig1 or sig2 ) // combinatorial logic
  @(posedge event1)       // wait for post edge transition of event1
  wait (signal  == 1)     // wait for signal to be 1
  ```

- Uncertainty
  Verilog supports some of the uncertainty inherent in electronic circuits. "X" is used to represent unknown state of the circuit. "Z" is used to represent undriven state of the circuit.

  ```
  reg1 = 1'bx;
  ```

```
reg2 = 1'bz;
```

- Abstraction
  Verilog supports designing at different levels of abstraction. The highest level of abstraction for a design is the Resister transfer Level (RTL), the next being the gate level and the lowest the cell level (User Define Primitives), RTL abstraction being the most commonly used. Verilog also supports behavioral level of abstraction with no regard to the structural realization of the design, primarily used for verification.

```
 // Example of a D flip flop at RTL abstraction
module dff (
 clk    , // Clock Input
 reset  , // Reset input
 d       , // Data Input
 q        // Q output
 );
 //-----------Input Ports---------------
 input d, clk, reset ;

 //-----------Output Ports---------------
 output q;

 reg q;

 always @ ( posedge clk)
 if (~reset) begin
   q <= 1'b0;
 end  else begin
   q <= d;
 end

endmodule


// And gate model based at Gate level abstraction
module and(input x,input y,output o);

wire  w;
// Two instantiations of the module NAND
nand U1(w,x, y);
nand U2(o, w, w);

endmodule

// Gate modeled at Cell-level Abstraction
primitive udp_and(
a, // declare three ports
b,
c
);
output a;   // Outputs
input b,c;  // Inputs

// UDP function code here
// A = B & C;
table
 // B  C    : A
    1  1    : 1;
```

```
    0  1    : 0;
    1  0    : 0;
    0  0    : 0;
endtable

endprimitive
```

There are three main use cases for Verilog. They determine the structure of the code and its interpretation and also determine the tool sets used. All three applications are necessary for successful implementation of any Verilog design.

1. Physical Design / Back-end
   Here Verilog is used to primarily view the design as a matrix of interconnecting gates implementing logical design. RTL/logic/Design goes through various steps from synthesis -> placement -> clock tree construction -> routing -> DRC -> LVS -> to tapeout. The precise steps and sequences vary based on the exact nature of implementation.
2. Simulation
   In this use case, the primary aim is to generate test vectors to validate the design as per the specification. The code written in this use case need not be synthesizable and it remains within verification sphere. The code here more closely resembles generic software structures like for/while/do loops etc.
3. Design
   Design involves implementing the specification of a circuit generally at the RTL level of abstraction. The Verilog code is then given for Verification and the fully verified code is given for physical implementation. The code is written using only the synthesizable constructs of Verilog. Certain RTL coding style can cause simulation vs synthesis mismatch and care has to be taken to avoid those.

There are two main implementation flows. They will also affect the way Verilog code is written and implemented. Certain styles of coding and certain structures are more suitable in one flow over the other.

- ASIC Flow (application-specific integrated circuit)
- FPGA Flow (Field-programmable gate array) - include FPGA and CPLD's

Hello World

This example uses the icarus verilog compiler.

Step 1: Create a file called hello.v

```
module myModule();

initial
  begin
    $display("Hello World!");   // This will display a message
    $finish ; // This causes the simulation to end.  Without, it would go on..and on.
  end
```

```
endmodule
```

Step 2. We compile the .v file using icarus:

```
>iverilog -o hello.vvp hello.v
```

The -o switch assigns a name to the output object file. Without this switch the output file would be called a.out. The hello.v indicates the source file to be compiled. There should be practically no output when you compile this source code, unless there are errors.

Step 3. You are ready to simulate this Hello World verilog program. To do so, invoke as such:

```
>vvp hello.vvp
Hello World!
>
```

Installation of Icarus Verilog Compiler for Mac OSX Sierra

1. Install Xcode from the App Store.
2. Install the Xcode developer tools

```
> xcode-select --install
```

This will provide basic command line tools such as `gcc` and `make`

3. Install Mac Ports `https://www.macports.org/install.php`

The OSX Sierra install package will provide an open-source method of installing and upgrading additional software packages on the Mac platform. Think `yum` or `apt-get` for the Mac.

4. Install icarus using Mac ports

```
> sudo port install iverilog
```

5. Verify the installation from the command line

```
$ iverilog
iverilog: no source files.

Usage: iverilog [-ESvV] [-B base] [-c cmdfile|-f cmdfile]
                [-g1995|-g2001|-g2005] [-g<feature>]
                [-D macro[=defn]] [-I includedir] [-M depfile] [-m module]
                [-N file] [-o filename] [-p flag=value]
                [-s topmodule] [-t target] [-T min|typ|max]
                [-W class] [-y dir] [-Y suf] source_file(s)

See the man page for details.
$
```

You are now ready to compile and simulate your first Verilog file on the Mac.

Install GTKWave for graphical display of simulation data on Mac OSx Sierra

GTKWave is a fully feature graphical viewing package that supports several graphical data storage standards, but it also happens to support VCD, which is the format that `vvp` will output. So, to pick up GTKWave, you have a couple options

1. Goto http://gtkwave.sourceforge.net/gtkwave.zip and download it. This version is typically the latest.
2. If you have installed MacPorts (https://www.macports.org/), simply run `sudo port install gtkwave`. This will probably want to install on the dependencies to. Note this method will usually get you an older version. If you don't have MacPorts installed, there is a installation setup example for doing this on this page. Yes! You will need all of the xcode developer tools, as this methods will "build" you a GTKWave from source.

When installation is done, you may be asked to select a python version. I already had 2.7.10 installed so I never "selected" a new one.

At this point you can start gtkwave from the command line with `gtkwave`. When it starts you may be asked to install, or update XQuarts. Do so. In my case XQuarts 2.7.11 is installed.

Note: I actually needed to reboot to get XQuarts correctly, then I typed `gtkwave` again and the application comes up.

In the next example, I will create two independent files, a testbench and module to test, and we will use the gtkwave to view the design.

Using Icarus Verilog and GTKWaves to simulate and view a design graphically

This example uses Icarus and GTKWave. Installation instructions for those tools on OSx are provided elsewhere on this page.

Lets begin with the module design. This module is a BCD to 7 segment display. I have coded the design in an obtuse way simply to give us something that is easily broken and we can spend sometime fixing graphically. So we have a clock, reset, a 4 data input representing a BCD value, and a 7 bit output that represent the seven segment display. Create a file called bcd_to_7seg.v and place the source below in it.

```
module bcd_to_7seg (
   input clk,
   input reset,
   input [3:0] bcd,
   output [6:0] seven_seg_display

);
   parameter TP = 1;
   reg seg_a;
   reg seg_b;
   reg seg_c;
   reg seg_d;
   reg seg_e;
   reg seg_f;
```

```
   reg seg_g;


   always @ (posedge clk or posedge reset)
      begin
      if (reset)
         begin
            seg_a <= #TP 1'b0;
            seg_b <= #TP 1'b0;
            seg_c <= #TP 1'b0;
            seg_d <= #TP 1'b0;
            seg_e <= #TP 1'b0;
            seg_f <= #TP 1'b0;
            seg_g <= #TP 1'b0;
         end
      else
         begin
            seg_a <= #TP  ~(bcd == 4'h1 || bcd == 4'h4);
            seg_b <= #TP  bcd < 4'h5 || bcd > 6;
            seg_c <= #TP   bcd != 2;
            seg_d <= #TP   bcd == 0 || bcd[3:1] == 3'b001 || bcd == 5 || bcd == 6 || bcd == 8;
            seg_e <= #TP  bcd == 0 || bcd == 2 || bcd == 6 || bcd == 8;
            seg_f <= #TP  bcd == 0 || bcd == 4 || bcd == 5 || bcd == 6 || bcd > 7;
            seg_g <= #TP  (bcd > 1 && bcd < 7) || (bcd > 7);
         end
    end

    assign seven_seg_display = {seg_g,seg_f,seg_e,seg_d,seg_c,seg_b,seg_a};
endmodule
```

Next, we need a test to check if this module is working correctly. The case statement in the testbench is actually easier to read in my opinion and more clear as to what it does. But I did not wan to put the same case statment in the design AND in the test. That is bad practice. Rather two independent designs are being used to validate one-another.

Withing the code below, you will notice two lines `$dumpfile("testbench.vcd");` and `$dumpvars(0,testbench);`. These lines are what create the VCD output file that will be used to perform graphical analysis of the design. If you leave them out, you won't get a VCD file generated. Create a file called testbench.v and place the source below in it.

```
`timescale 1ns/100ps
module testbench;
reg clk;
reg reset;
reg [31:0] ii;
reg [31:0] error_count;
reg [3:0] bcd;
wire [6:0] seven_seg_display;
parameter TP = 1;
parameter CLK_HALF_PERIOD = 5;

   // assign clk = #CLK_HALF_PERIOD ~clk;  // Create a clock with a period of ten ns
   initial
   begin
     clk = 0;
     #5;
     forever clk = #( CLK_HALF_PERIOD )  ~clk;
   end
```

```
   initial
     begin
       $dumpfile("testbench.vcd");
       $dumpvars(0,testbench);
       // clk  = #CLK_HALF_PERIOD ~clk;
       $display("%0t, Reseting system", $time);
       error_count = 0;
       bcd  = 4'h0;
       reset = #TP 1'b1;
       repeat (30) @ (posedge clk);
       reset  = #TP 1'b0;
       repeat (30) @ (posedge clk);
       $display("%0t, Begin BCD test", $time); // This displays a message


       for (ii = 0; ii < 10; ii = ii + 1)
          begin
          repeat (1) @ (posedge clk);
          bcd  = ii[3:0];
          repeat (1) @ (posedge clk);
          if (seven_seg_display !== seven_seg_prediction(bcd))
             begin
                $display("%0t, ERROR: For BCD %d, module output 0b%07b does not match
prediction logic value of 0b%07b.",$time,bcd, seven_seg_display,seven_seg_prediction(bcd));
                error_count = error_count + 1;
             end
          end
       $display("%0t, Test Complete with %d errors", $time, error_count);
       $display("%0t, Test %s", $time, ~|error_count ? "pass." : "fail.");
       $finish ; // This causes the simulation to end.
     end


parameter SEG_A = 7'b0000001;
parameter SEG_B = 7'b0000010;
parameter SEG_C = 7'b0000100;
parameter SEG_D = 7'b0001000;
parameter SEG_E = 7'b0010000;
parameter SEG_F = 7'b0100000;
parameter SEG_G = 7'b1000000;

function [6:0] seven_seg_prediction;
   input [3:0] bcd_in;

   //    +--- A ---+
   //    |         |
   //    F         B
   //    |         |
   //    +--- G ---+
   //    |         |
   //    E         C
   //    |         |
   //    +--- D ---+

   begin
      case (bcd_in)
         4'h0: seven_seg_prediction = SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F;
         4'h1: seven_seg_prediction = SEG_B | SEG_C;
         4'h2: seven_seg_prediction = SEG_A | SEG_B | SEG_G | SEG_E | SEG_D;
         4'h3: seven_seg_prediction = SEG_A | SEG_B | SEG_G | SEG_C | SEG_D;
```

```
         4'h4: seven_seg_prediction = SEG_F | SEG_G | SEG_B | SEG_C;
         4'h5: seven_seg_prediction = SEG_A | SEG_F | SEG_G | SEG_C | SEG_D;
         4'h6: seven_seg_prediction = SEG_A | SEG_F | SEG_G | SEG_E | SEG_C | SEG_D;
         4'h7: seven_seg_prediction = SEG_A | SEG_B | SEG_C;
         4'h8: seven_seg_prediction = SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F | SEG_G;
         4'h9: seven_seg_prediction = SEG_A | SEG_F | SEG_G | SEG_B | SEG_C;
         default: seven_seg_prediction = 7'h0;
      endcase
   end
endfunction


bcd_to_7seg u0_bcd_to_7seg (
.clk               (clk),
.reset             (reset),
.bcd               (bcd),
.seven_seg_display (seven_seg_display)
);


endmodule
```

Now that we have two files, a testbench.v and bcd_to_7seg.v, we need to compile, elaborate using Icarus. To do this:

```
$ iverilog -o testbench.vvp testbench.v bcd_to_7seg.v
```

Next we need to simulate

```
$ vvp testbench.vvp
LXT2 info: dumpfile testbench.vcd opened for output.
0, Reseting system
6000, Begin BCD test
8000, Test Complete with          0 errors
8000, Test pass.
```

At this point if you want to validate the file is really being tested, go into the bcd_2_7seg.v file and move some of the logic around and repeat those first two steps.

As an example I change the line `seg_c <= #TP bcd != 2;` to `seg_c <= #TP bcd != 4;`. Recompile and simulate does the following:

```
$ iverilog -o testbench.vvp testbench.v bcd_to_7seg.v
$ vvp testbench.vvp
LXT2 info: dumpfile testbench.vcd opened for output.
0, Reseting system
6000, Begin BCD test
6600, ERROR: For BCD  2, module output 0b1011111 does not match prediction logic value of
0b1011011.
7000, ERROR: For BCD  4, module output 0b1100010 does not match prediction logic value of
0b1100110.
8000, Test Complete with          2 errors
8000, Test fail.
$
```

So now, lets view the simulation using GTKWave. From the command line, issue a

```
gtkwave testbench.vcd &
```

When the GTKWave window appears, you will see in the upper left hand box, the module name testbench. Click it. This will reveal the sub-modules, tasks, and functions associated with that file. Wires and registers will also appear in the lower left hand box.

Now drag, clk, bcd, error_count and seven_seg_display into the signal box next to the waveform window. The signals will now be plotted. Error_count will show you which particular BCD input generated the wrong seven_seg_display output.

You are now ready to troubleshoot a Verilog bug graphically.

Read Getting started with verilog online: https://riptutorial.com/verilog/topic/1080/getting-started-with-verilog

Chapter 2: Hello World

Examples

Compiling and Running the Example

Assuming a source file of `hello_world.v` and a top level module of `hello_world`. The code can be run using various simulators. Most simulators are compiled simulators. They require multiple steps to compile and execute. Generally the

- First step is to compile the Verilog design.
- Second step is to elaborate and optimize the design.
- Third step is to run the simulation.

The details of the steps could vary based on the simulator but the overall idea remains the same.

Three step process using Cadence Simulator

```
ncvlog hello_world.v
ncelab hello_world
ncsim hello_world
```

- First step ncvlog is to compile the file hello_world.v
- Second step ncelab is to elaborate the code with the top level module hello_world.
- Third step ncsim is to run the simulation with the top level module hello_world.
- The simulator generates all the compiled and optimized code into a work lib. [INCA_libs - default library name]

single step using Cadence Simulator.

The command line will internally call the required three steps. This is to mimic the older interpreted simulator execution style (single command line).

```
irun hello_world.v
or
ncverilog hello_world.v
```

Hello World

The program outputs Hello World! to standard output.

```
module HELLO_WORLD(); // module doesn't have input or outputs
  initial begin
    $display("Hello World");
    $finish; // stop the simulator
  end
endmodule
```

Module is a basic building block in Verilog. It represent a collection of elements and is enclosed between module and end module keyword. Here hello_world is the top most (and the only) module .

Initial block executes at the start of simulation. The begin and end is used to mark the boundary of the initial block. `$display` outputs the message to the standard output. It inserts and end of line "\n" to the message.

This code can't by synthesized, i.e. it can't be put in a chip.

Read Hello World online: https://riptutorial.com/verilog/topic/1819/hello-world

Chapter 3: Memories

Remarks

For FIFOs, you typically instantiate a vendor-specific block (also called a "core" or "IP").

Examples

Simple Dual Port RAM

Simple Dual Port RAM with separate addresses and clocks for read/write operations.

```
module simple_ram_dual_clock #(
  parameter DATA_WIDTH=8,                 //width of data bus
  parameter ADDR_WIDTH=8                  //width of addresses buses
)(
  input      [DATA_WIDTH-1:0] data,       //data to be written
  input      [ADDR_WIDTH-1:0] read_addr,  //address for read operation
  input      [ADDR_WIDTH-1:0] write_addr, //address for write operation
  input                       we,         //write enable signal
  input                       read_clk,   //clock signal for read operation
  input                       write_clk,  //clock signal for write operation
  output reg [DATA_WIDTH-1:0] q           //read data
);

  reg [DATA_WIDTH-1:0] ram [2**ADDR_WIDTH-1:0]; // ** is exponentiation

  always @(posedge write_clk) begin //WRITE
    if (we) begin
      ram[write_addr] <= data;
    end
  end

  always @(posedge read_clk) begin //READ
    q <= ram[read_addr];
  end

endmodule
```

Single Port Synchronous RAM

Simple Single Port RAM with one address for read/write operations.

```
module ram_single #(
  parameter DATA_WIDTH=8,           //width of data bus
  parameter ADDR_WIDTH=8            //width of addresses buses
)(
  input  [(DATA_WIDTH-1):0] data,  //data to be written
  input  [(ADDR_WIDTH-1):0] addr,  //address for write/read operation
  input                     we,    //write enable signal
  input                     clk,   //clock signal
  output [(DATA_WIDTH-1):0] q      //read data
```

```
);

  reg [DATA_WIDTH-1:0] ram [2**ADDR_WIDTH-1:0];
  reg [ADDR_WIDTH-1:0] addr_r;

  always @(posedge clk) begin //WRITE
      if (we) begin
          ram[addr] <= data;
      end
      addr_r <= addr;
  end

  assign q = ram[addr_r]; //READ

endmodule
```

Shift register

N-bit deep shift register with asynchronous reset.

```
module shift_register #(
  parameter REG_DEPTH = 16
)(
  input clk,       //clock signal
  input ena,       //enable signal
  input rst,       //reset signal
  input data_in,   //input bit
  output data_out  //output bit
);

  reg [REG_DEPTH-1:0] data_reg;

  always @(posedge clk or posedge rst) begin
    if (rst) begin //asynchronous reset
      data_reg <= {REG_DEPTH{1'b0}}; //load REG_DEPTH zeros
    end else if (enable) begin
      data_reg <= {data_reg[REG_DEPTH-2:0], data_in}; //load input data as LSB and shift
(left) all other bits
    end
  end

  assign data_out = data_reg[REG_DEPTH-1]; //MSB is an output bit

endmodule
```

Single Port Async Read/Write RAM

Simple single port RAM with async read/write operations

```
module ram_single_port_ar_aw #(
  parameter DATA_WIDTH = 8,
  parameter ADDR_WITDH = 3
)(
  input                       we,    // write enable
  input                       oe,    // output enable
  input  [(ADDR_WITDH-1):0]   waddr, // write address
  input  [(DATA_WIDTH-1):0]   wdata, // write data
```

```
  input                         raddr, // read adddress
  output [(DATA_WIDTH-1):0]     rdata  // read data
);

  reg [(DATA_WIDTH-1):0]       ram [0:2**ADDR_WITDH-1];
  reg [(DATA_WIDTH-1):0]       data_out;

  assign rdata = (oe && !we) ? data_out : {DATA_WIDTH{1'bz}};

  always @*
  begin : mem_write
    if (we) begin
      ram[waddr] = wdata;
    end
  end

  always @* // if anything below changes (i.e. we, oe, raddr), execute this
  begin : mem_read
    if (!we && oe) begin
      data_out = ram[raddr];
    end
  end

endmodule
```

Read Memories online: https://riptutorial.com/verilog/topic/2654/memories

Chapter 4: Procedural Blocks

Syntax

- always @ (posedge clk) begin /* statements */ end
- always @ (negedge clk) begin /* statements */ end
- always @ (posedge clk or posedge reset) // may synthesize less efficiently than synchronous reset

Examples

Simple counter

A counter using an FPGA style flip-flop initialisation:

```
module counter(
    input clk,
    output reg[7:0] count
)
initial count = 0;
always @ (posedge clk) begin
    count <= count + 1'b1;
end
```

A counter implemented using asynchronous resets suitable for ASIC synthesis:

```
module counter(
  input            clk,
  input            rst_n, // Active-low reset
  output reg [7:0] count
)
always @ (posedge clk or negedge rst_n) begin
  if (~rst_n) begin
    count <= 'b0;
  end
  else begin
    count <= count + 1'b1;
  end
end
```

The procedural blocks in these examples increment `count` at every rising clock edge.

Non-blocking assignments

A non-blocking assignment (`<=`) is used for assignment inside edge-sensitive `always` blocks. Within a block, the new values are not visible until the entire block has been processed. For example:

```
module flip(
    input clk,
```

```
    input reset
)
reg f1;
reg f2;

always @ (posedge clk) begin
  if (reset) begin // synchronous reset
    f1 <= 0;
    f2 <= 1;
  end
  else begin
    f1 <= f2;
    f2 <= f1;
  end
end
endmodule
```

Notice the use of non-blocking (`<=`) assignments here. Since the first assignment doesn't actually take effect until after the procedural block, the second assignment does what is intended and actually swaps the two variables -- unlike in a blocking assignment (`=`) or assignments in other languages; `f1` still has its original value on the right-hand-side of the second assignment in the block.

Read Procedural Blocks online: https://riptutorial.com/verilog/topic/2512/procedural-blocks

Chapter 5: Synthesis vs Simulation mismatch

Introduction

A good explanation of this topic is in http://www.sunburst-design.com/papers/CummingsSNUG1999SJ_SynthMismatch.pdf

Examples

Comparison

wire d = 1'bx; // say from previous block. Will be 1 or 0 in hardware

if (d == 1'b) // false in simulation. May be true of false in hardware

Sensitivity list

```
wire a;
wire b;
reg q;

always @(a) // b missing from sensativity list
 q = a & b; // In simulation q will change only when a changes
```

In hardware, q will change whenever a or b changes.

Read Synthesis vs Simulation mismatch online: https://riptutorial.com/verilog/topic/9220/synthesis-vs-simulation-mismatch

Credits

S. No	Chapters	Contributors
1	Getting started with verilog	Brian Carlton, Community, hexafraction, Morgan, Qiu, Rahul Menon, Rich Maes, wilcroft
2	Hello World	Brian Carlton, Morgan, Rahul Menon
3	Memories	Brian Carlton, jclin, Kamil Rymarz, Morgan, Qiu, wilcroft
4	Procedural Blocks	Brian Carlton, hexafraction, Morgan, wilcroft
5	Synthesis vs Simulation mismatch	Brian Carlton