Using Verilog for Testbenches

Design of Digital Circuits 2014
Srdjan Capkun
Frank K. Gürkaynak

http://www.syssec.ethz.ch/education/Digitaltechnik_14
What Will We Learn?

- How to simulate your circuit
- Applying inputs
- Seeing if the circuit does the correct thing
How Do You Know That A Circuit Works?

- You have written the Verilog code of a circuit
  - Does it work correctly?
  - Even if the syntax is correct, it might do what you want?
  - What exactly it is that you want anyway?

- Trial and error can be costly
  - You need to ‘test’ your circuit in advance

- In modern digital designs, functional verification is the most time consuming design stage.
The Idea Behind A Testbench

- Using a computer simulator to test your circuit
  - You instantiate your design
  - Supply the circuit with some inputs
  - See what it does
  - Does it return the “correct” outputs?
Testbenches

- HDL code written to test another HDL module, the *device under test* (dut), also called the *unit under test* (uut)

- Not synthesizeable

- Types of testbenches:
  - Simple testbench
  - Self-checking testbench
  - Self-checking testbench with testvectors
Example

- Write Verilog code to implement the following function in hardware:

  \[ y = (\overline{b} \cdot \overline{c}) + (a \cdot \overline{b}) \]

- Name the module sillyfunction
Example

- Write Verilog code to implement the following function in hardware:

\[ y = (\overline{b} \cdot \overline{c}) + (a \cdot \overline{b}) \]

- Name the module `sillyfunction`

```verilog
module sillyfunction(input a, b, c, 
                     output y);

    assign y = ~b & ~c | a & ~b;
endmodule
```
module testbench1(); // Testbench has no inputs, outputs
  reg a, b, c; // Will be assigned in initial block
  wire y;

  // instantiate device under test
  sillyfunction dut (.a(a), .b(b), .c(c), .y(y));

  // apply inputs one at a time
  initial begin // sequential block
      a = 0; b = 0; c = 0; #10; // apply inputs, wait 10ns
      c = 1; #10; // apply inputs, wait 10ns
      b = 1; c = 0; #10; // etc .. etc..
      c = 1; #10;
      a = 1; b = 0; c = 0; #10;
  end
endmodule
Simple Testbench

- Simple testbench instantiates the design under test
- It applies a series of inputs
- The outputs have to be observed and compared using a simulator program.
  - This type of testbench does not help with the outputs
- `initial` statement is similar to `always`, it just starts once at the beginning, and does not repeat.
- The statements have to be blocking.
Self-checking Testbench

```verilog
module testbench2();
    reg   a, b, c;
    wire  y;

    // instantiate device under test
    sillyfunction dut(.a(a), .b(b), .c(c), .y(y));

    // apply inputs one at a time
    initial begin
        a = 0; b = 0; c = 0; #10; // apply input, wait
        if (y !== 1) $display("000 failed."); // check
        c = 1; #10; // apply input, wait
        if (y !== 0) $display("001 failed."); // check
        b = 1; c = 0; #10; // etc.. etc..
        if (y !== 0) $display("010 failed."); // check
    end
endmodule
```
Self-checking Testbench

- Better than simple testbench
- This testbench also includes a statement to check the current state
- `$display` will write a message in the simulator
- This is a lot of work
  - Imagine a 32-bit processor executing a program (thousands of clock cycles)
- You make the same amount of mistakes when writing testbenches as you do writing actual code
Testbench with Testvectors

- The more elaborate testbench

- Write testvector file: inputs and expected outputs
  - Usually can use a high-level model (golden model) to produce the ‘correct’ input output vectors

- Testbench:
  - Generate clock for assigning inputs, reading outputs
  - Read testvectors file into array
  - Assign inputs, get expected outputs from DUT
  - Compare outputs to expected outputs and report errors
Testbench with Testvectors

- **A testbench clock is used to synchronize I/O**
  - The same clock can be used for the DUT clock

- **Inputs are applied following a hold margin**

- **Outputs are sampled before the next clock edge**
  - The example in book uses the falling clock edge to sample
Testvectors File

- We need to generate a testvector file (somehow)

- File: example.tv – contains vectors of abc_yexpected

  000_1
  001_0
  010_0
  011_0
  100_1
  101_1
  110_0
  111_0
Testbench: 1. Generate Clock

```verilog
module testbench3();
    reg clk, reset;       // clock and reset are internal
    reg a, b, c, yexpected;   // values from testvectors
    wire y;                // output of circuit
    reg [31:0] vectornum, errors;   // bookkeeping variables
    reg [3:0] testvectors[10000:0];  // array of testvectors

    // instantiate device under test
    sillyfunction dut(.a(a), .b(b), .c(c), .y(y));

    // generate clock
    always // no sensitivity list, so it always executes
        begin
            clk = 1; #5; clk = 0; #5;       // 10ns period
        end
```
2. Read Testvectors into Array

// at start of test, load vectors
// and pulse reset

initial // Will execute at the beginning once
begin
  $readmemb("example.tv", testvectors); // Read vectors
  vectornum = 0; errors = 0; // Initialize
  reset = 1; #27; reset = 0; // Apply reset wait
end

// Note: $readmemh reads testvector files written in
// hexadecimal
3. Assign Inputs and Expected Outputs

// apply test vectors on rising edge of clk
always @(posedge clk)
begin
    #1; {a, b, c, yexpected} = testvectors[vectornum];
end

- Apply inputs with some delay (1ns) AFTER clock

- This is important
  - Inputs should not change at the same time with clock

- Ideal circuits (HDL code) are immune, but real circuits (netlists) may suffer from hold violations.
4. Compare Outputs with Expected Outputs

```verilog
// check results on falling edge of clk
always @(negedge clk)
  if (~reset) // skip during reset
    begin
      if (y !== yexpected)
        begin
          $display("Error: inputs = %b", {a, b, c});
          $display(" outputs = %b (%b exp)", y, yexpected);
          errors = errors + 1;
        end
    end
// Note: to print in hexadecimal, use %h. For example,
// $display("Error: inputs = %h", {a, b, c});
```
4. Compare Outputs with Expected Outputs

// increment array index and read next test vector
vectornum = vectornum + 1;
if (testvectors[vectornum] === 4'bx)
begin
    $display("%d tests completed with %d errors", vectornum, errors);
    $finish; // End simulation
end
end
endmodule

// Note: === and !== can compare values that are
// x or z.
Golden Models

- A golden model represents the ideal behavior of your circuit.
  - Still it has to be developed
  - It is difficult to get it right (bugs in the golden model!)
  - Can be done in C, Perl, Python, Matlab or even in Verilog

- The behavior of the circuit is compared against this golden model.
  - Allows automated systems (very important)
Why is Verification difficult?

- **How long would it take to test a 32-bit adder?**
  - In such an adder there are 64 inputs = $2^{64}$ possible inputs
  - That makes around $1.85\times10^{19}$ possibilities
  - If you test one input in 1ns, you can test $10^9$ inputs per second
    - or $8.64\times10^{14}$ inputs per day
    - or $3.15\times10^{17}$ inputs per year
  - We would still need **58.5 years** to test all possibilities

- **Brute force testing is not feasible for all circuits, we need alternatives**
  - Formal verification methods
  - Choosing ‘critical cases’
  - Not an easy task
What did we learn?

- Verilog has other uses than modeling hardware
  - It can be used for creating testbenches

- Three main classes of testbenches
  - Applying only inputs, manual observation (*not a good idea*)
  - Applying and checking results with inline code (*cumbersome*)
  - Using testvector files (*good for automatization*)