

Simulation

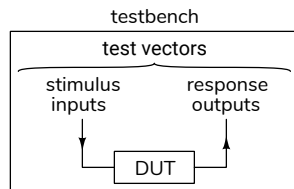
This lecture describes how to use Verilog to simulate designs.

After this lecture you should be able to write a testbench that can: set initial values, generate clocks, read test vectors from a file, display values, and terminate on a condition.

Version 2: corrected testbench.

Simulation

Verilog can be used to test HDL designs by simulating their operation. A simulation consists of the module being tested (called the Design Under Test or DUT) that is instantiated in another module called a testbench. The testbench applies inputs to the DUT and checks its outputs:



Test Vectors

The inputs to the DUT and the corresponding expected outputs are called test vectors. These can be generated by the testbench itself or they can be read from a file.

Test vectors should include:

1. typical inputs,
2. minimum and maximum valid inputs,
3. invalid inputs, and
4. randomly-chosen values.

Exercise 1: Give examples of appropriate test inputs for each of the above categories if you were testing a circuit that computed the square root of a 16-bit signed number.

Verilog for Verification

The following Verilog features are useful for simulation but can't be synthesized (implemented in hardware).

initial and always blocks

initial blocks execute once at the start of the simulation and are used to initialise signals. **always** blocks execute continuously. **begin** and **end** are used to group statements in these blocks (as with **{** and **}** in C).

if/else/for/while

if/else, **for**, and **while** statements, whose syntax is similar to C, can be used in **initial** or **always** blocks. However, the ability to execute statements based on conditions or events reduces the need for these statements.

Delays

Placing **#number** before a statement delays¹ execution by **number** simulation time. The suffixes **ns** and **us** can be used for nano- and micro-seconds.

The syntax **@(event)** where **event** can be **posedge** or **negedge** before a signal name or just a signal name delays execution until that signal edge or a change in that signal value.

wait

The **wait(expression)** statement pauses until the expression is non-zero.

Exercise 2: What's the difference between **wait(x) y='1;** and **@(x) y='1;?**

System Tasks

Functions beginning with **\$** are called system tasks. Useful ones include:

¹Delays are not synthesizable because delays cannot be easily implemented in hardware.

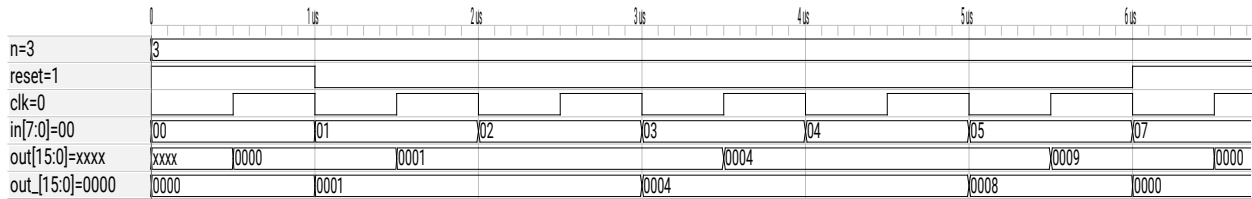


Figure 1: Simulation waveforms (from `ex66.vcd`).

- `$display()` similar to C's `printf()`, can be used to print values during a simulation;
- `$dumpfile` and `$dumpvars` record changes in signals to a `.vcd` file for subsequent viewing with a waveform viewer.
- `$fopen()` and `$fscanf()`, similar to the C library functions `fopen()` and `fscanf()`, can open and read from text files.
- `$finish` and `$stop` terminate or suspend a simulation.

```

always_ff @(posedge clk)
    out <= reset ? '0 :
        in & 1 ? out+in : out ;

endmodule

// example testbench

module ex66_tb ;

    // DUT inputs and outputs
    logic reset, clk ;
    logic [7:0] in ;
    logic [15:0] out, out_ ;

    // file descriptor and values read per fscanff()
    integer fd, n ;

    // instantiate DUT
    ex66 ex66_0 (.*) ;

    // initialization
    initial begin

        // record all signals in a .vcd file
        $dumpfile("ex66.vcd") ;
        $dumpvars ;

        // initialize clock
        clk = '0 ;

        // open the test vector file
        fd = $fopen("ex66data.csv", "r") ;

        // wait for end of file or error
        wait (n < 0) $stop ;
    end

    // generate 1 MHz clock (2x500ns period)
    always #0.5us clk = ~clk ;

    // read/apply/check test vectors
    always begin
        // read & set DUT inputs; display test vector
        n = $fscanf(fd, "%d,%d,%d", reset, in, out_) ;
        $display ("test vector=%d,%d,%d", reset, in, out_) ;

        // on falling clock edge
        @(negedge clk) begin
            // display DUT internal signals; check outputs
            $display ( "DUT state = %d", ex66_0.out ) ;
            if ( out != out_ ) begin
                $display("***Error: %d, %d, %d", in, out, out_);
            end
        end
    end
endmodule

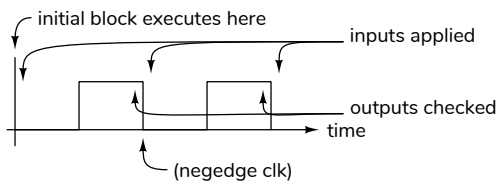
```

Example

The testbench below demonstrates the language features described above.

The DUT is a module with an 8-bit input that outputs a 16-bit sum of all the odd-valued inputs since a reset input was asserted. An `initial` block opens a file containing test vectors, initializes the clock signal and `$stop`'s the simulation at the end of the test vector file. Every $0.5 \mu\text{s}$ the clock is inverted; this creates a 1 MHz clock.

The final `always` block continuously reads and applies the input(s) from each test vector, waits for the falling edge of the clock, and compare the DUT output(s) to the value(s) read from the test vectors:



```
// output cumulative sum of odd-valued inputs
```

```

module ex66
    ( input logic reset, clk,
      input logic [7:0] in,
      output logic [15:0] out ) ;

```

```
        end
    end

endmodule
```

The test vectors are read from the file `ex66data.csv` containing the following lines²:

```
1,0,0
0,1,1
0,2,1
0,3,4
0,4,4
0,5,8
1,7,0
```

Running this testbench using the Modelsim simulation program creates the waveform files shown in Figure 1 and, since the test vector contains an error, the following line is printed showing the input, actual output and the expected output given in the test vector:

```
# test vector=1, 0, 0
# DUT state = 0
# test vector=0, 1, 1
# DUT state = 1
# test vector=0, 2, 1
# DUT state = 1
# test vector=0, 3, 4
# DUT state = 4
# test vector=0, 4, 4
# DUT state = 4
# test vector=0, 5, 8
# DUT state = 9
# ***Error: 5, 9, 8
# test vector=1, 7, 0
# DUT state = 0
# test vector=1, 7, 0
```

Exercise 3: How could you:

- terminate the simulation if a test vector failed?
- change the clock frequency to 10 MHz?
- print each test vector as it's read?
- assert the reset input for two clock cycles?

²This file is in Comma Separated Values (.csv) format.