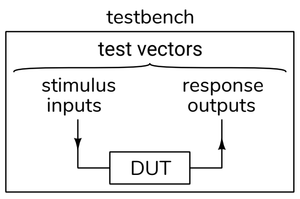
Simulation

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.

Simulation

In addition to using HDLs to design hardware, they can also be used to test these 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 within 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, in HDL simulations control flow is often based on conditions or events as described below.

Delays

Placing #*number* before a statement delays[1] 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`** followed by a signal name or just the signal name delays execution until that signal edge or until a change in that signal's 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:

- `$write()` is similar to C's `printf()`, and can be used to print values during a simulation;

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

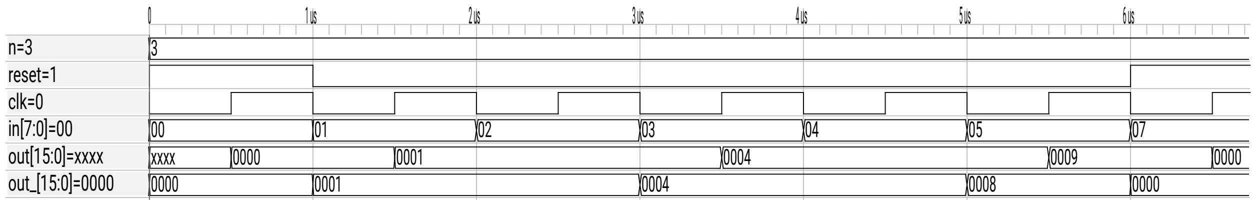


Figure 1: Simulation waveforms (from **ex66.vcd**).

- **$dumpfile** and **$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.

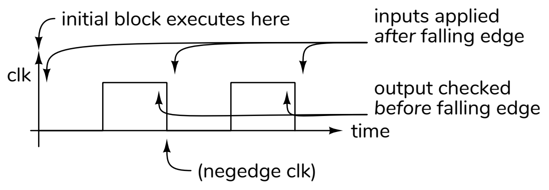
Example

The testbench below demonstrates the language features described above.

The example 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 µs 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 compares the DUT output(s) to the value(s) read from the test vectors.

The example testbench checks the DUT outputs just before the falling edge of the clock and changes the DUT inputs just after the falling edge:



This offsets changes in the inputs and outputs by half a clock period from the rising clock edge to make the waveforms easier to understand. This is a *functional* simulation which means the delay of the DUT is not being checked[2]

Note that the simulation code accesses the value of **out** within the DUT using the notation <*instance name*>.<*signal name*> (**ex66_0.out** in this case). This violates the principles of modularity and "data hiding" but can be useful for troubleshooting.

The test vectors are read from the file **ex66data.csv** containing the following lines[3]:

```
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. The following lines are printed showing the inputs, actual output, and the expected output:

```
# run -all
#     reset      in     out    out_
#         1       0       0       0
#         0       1       1       1
#         0       2       1       1
#         0       3       4       4
#         0       4       4       4
#         0       5       8       9 *** Error
#         1       7       0       0
# ** Note: $stop    : ex66.sv(55)
#    Time: 7 us  Iteration: 1  Instance: /ex66_tb
# Break in Module ex66_tb at ex66.sv line 55
```

Exercise 3: How could you:

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

[2]If the DUT description includes delays then a simulation can also check that the DUT operates correctly at the applied clock rate.

[3]This file is in Comma Separated Values (.csv) format.

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

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

   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 number of values by fscanf()
   integer fd, n ;

   // instantiate DUT
   ex66 ex66_0 (.*) ;

   // initialization
   initial begin

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

      // initialize variables and open test vector file
      n = 3 ;
      clk = '0 ;
      fd = $fopen("ex66data.csv","r") ;

      $write("%8s%8s%8s%8s\n",
             "reset", "in", "out", "out_") ;
   end

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

   // continuously apply input, wait, & check output
   always begin

      // read (and apply) DUT inputs
      n = $fscanf(fd,"%d,%d,%d", reset, in, out_) ;

      // stop if didn't read 3 values
      if ( n != 3 ) $stop ;

      //  display test vector
      $write("%8d%8d%8d",reset,in,out_) ;

      // wait for falling clock edge
      @(negedge clk) ;

      // show DUT state and check output
      $write( "%8d%s\n", ex66_0.out,
              out != out_ ? " *** Error" : "" ) ;

   end

endmodule
```