Chapter E

Online Appendix: Creating Verilog Testbenches

Once you have written a Verilog design file, you may want to simulate it to see if the design behaves as you expect. The WebFPGA IDE does not support simulation but many HDL development environments do. Here, we will use `https://LAoE.link/EDA_Playground.html` to demonstrate Verilog simulation. There are respectable arguments *against* simulating. For the simple designs we ask you to develop, it may take more time to create and debug the simulation program than to just debug your code through trial-and-error. Indeed, you may ignore this chapter and successfully get through the programmable logic designs we ask of you, but if you are curious how we test our students' homework assignments, read on.

E.1 The form of a Verilog file: simulation testbench

Simulation files are difficult to write, and for a complex design often cannot be exhaustive: you cannot try every possible case. This is especially likely for sequential designs.

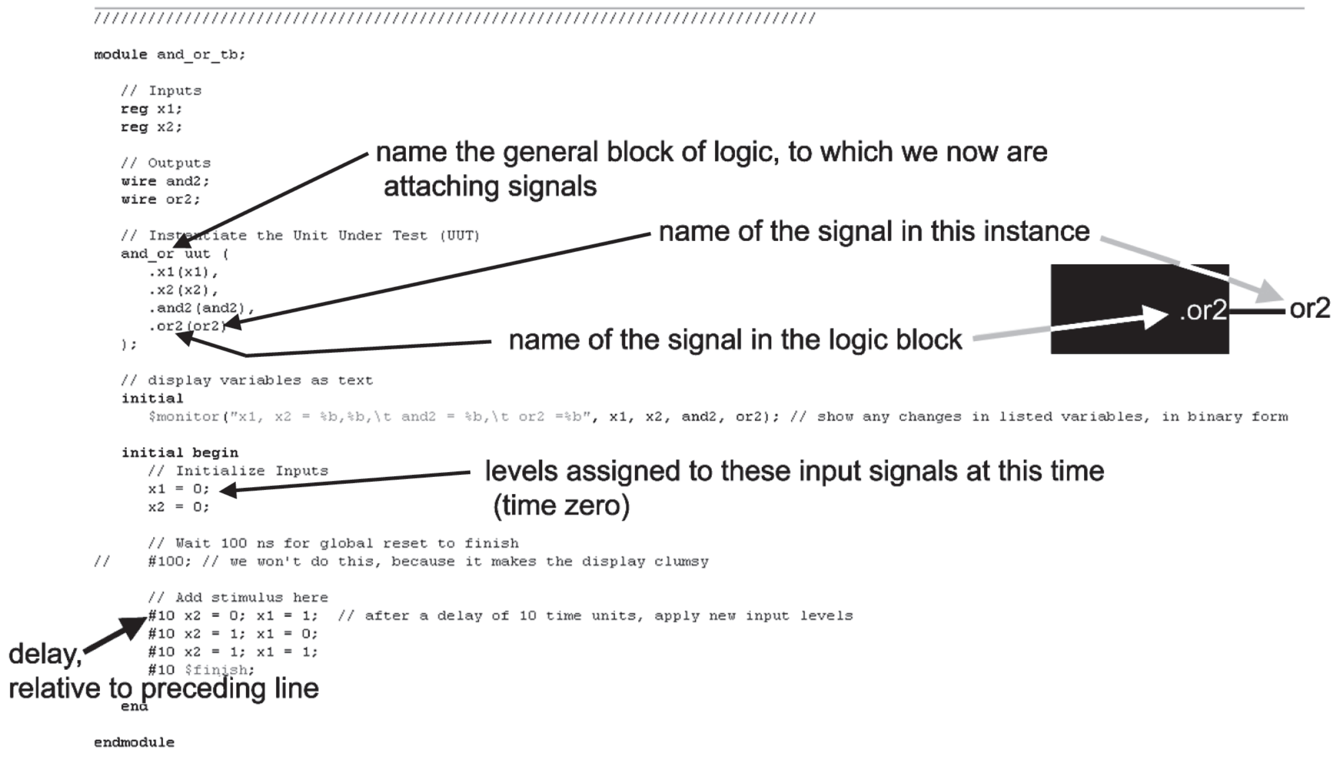
The simulation file, called a testbench, is a Verilog text file that exercises a design.[1] In the case of an AND and an OR gate this is hardly necessary, but the extreme simplicity of the AND–OR logic should make the testbench easy to follow.

The testbench begins something like the design file, listing inputs and output variables, and stating their types – wire or reg. Just in case you were thinking you're beginning to understand the wire/reg distinction, note that the *inputs* in the simulation file are always of type *reg*, the outputs are *wire*. The wire/reg distinction is difficult. The type reg bears a ghostly relation to the hardware element, "register," but only ghostly. A variable of type *reg* has a value "retained in memory…until changed by a subsequent assignment."[2] That

[1] Strictly, Verilog calls such a test file "Test Fixture." Testbench is VHDL jargon, but we use it because it seems to be more widely used than test fixture.

[2] J. Cavanagh, *Verilog HDL* (2007), p. 77.

makes it sound like a collection of flip-flops, but it emphatically is not that, and does not imply use of flip-flops. Fig. E.1 shows a testbench for exercising the AND–OR logic.[3]



```
////////////////////////////////////////////////////////////////////////////////

module and_or_tb;

   // Inputs
   reg x1;
   reg x2;

   // Outputs
   wire and2;
   wire or2;

   // Instantiate the Unit Under Test (UUT)
   and_or uut (
      .x1(x1),
      .x2(x2),
      .and2(and2),
      .or2(or2)
   );

   // display variables as text
   initial
      $monitor("x1, x2 = %b,%b,\t and2 = %b,\t or2 =%b", x1, x2, and2, or2); // show any changes in listed variables, in binary form

   initial begin
      // Initialize Inputs
      x1 = 0;
      x2 = 0;

      // Wait 100 ns for global reset to finish
//    #100; // we won't do this, because it makes the display clumsy

      // Add stimulus here
      #10 x2 = 0; x1 = 1;  // after a delay of 10 time units, apply new input levels
      #10 x2 = 1; x1 = 0;
      #10 x2 = 1; x1 = 1;
      #10 $finish;
   end

endmodule
```

name the general block of logic, to which we now are attaching signals
name of the signal in this instance
name of the signal in the logic block
.or2
or2
levels assigned to these input signals at this time (time zero)
delay, relative to preceding line

Figure E.1: Verilog testbench.

Simulation results: The result, in Fig. E.2, of simulation is what one would expect. OR looks like an OR of x1, x2; AND looks like an AND.

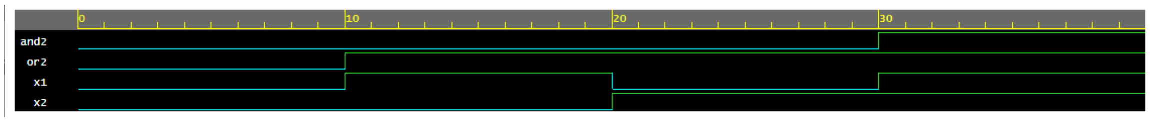


Figure E.2: Waveform output from Verilog testbench.

Instantiation: Some elements of the testbench require explanation. One is the "instantiation" block:

```
// Instantiate the Unit Under Test (UUT)
and_or uut (
```

[3]You can try this simulation at https://LAoE.link/AND_OR_Simulation.html. We have added the two commands after the comment "`// Dump waves`" so the EPWave application can display the output graphically.

```
        .x1(x1),
        .x2(x2),
        .and2(and2),
        .or2(or2)
    );
```

Instantiate is a weird word for a less weird concept. It means that it takes the defined logic block (in this case, a block providing an AND and an OR gate) and uses it to process signals in a particular *instance*. The word is chosen to indicate that we are coming down from the abstraction of a high level design into an "instance" that make the design real.

The above code (`.and2(and2)`) indicates what we're doing in the present case: the mundane operation of linking the particular name in this instance ("and2") with the name of the variable coming out of the logic block (`.and2`).

But the fact that the two variables carry almost the same name – one from the design block, and another from the particular instance – obscures the fact that they are quite different, independent notions. To make that point, we've made a second testbench with sillier names. Fig. E.3 is a sketch that tries to make the point that the and_or logic block's inputs and outputs can be named as we please, when the block is invoked by a particular use or instantiation.

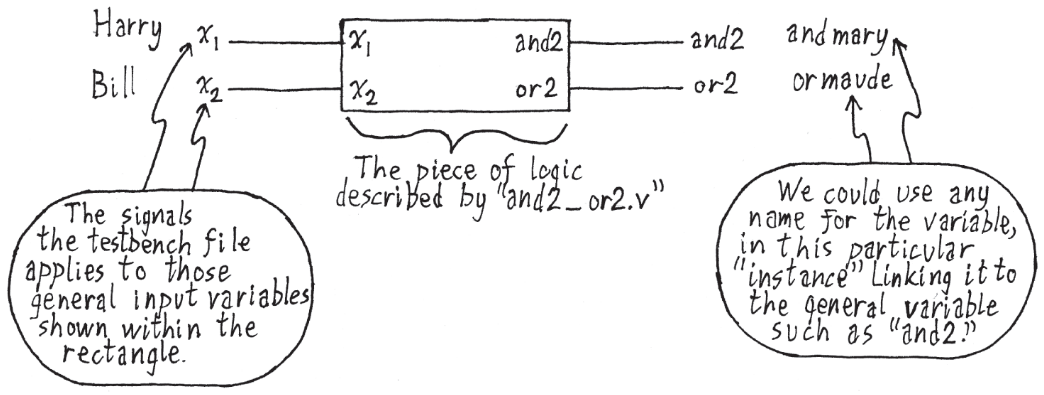
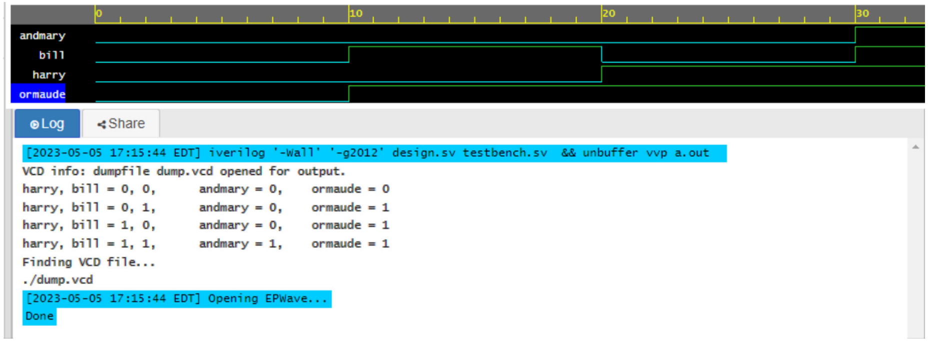

Harry
Bill
x_1
x_2
x_1
x_2
and2
or2
and2
or2
and mary
or maude
The piece of logic described by "and2_or2.v"
The signals the testbench file applies to those general input variables shown within the rectangle.
We could use any name for the variable, in this particular "instance" linking it to the general variable such as "and2."

Figure E.3: Instantiation illustrated: the underlying logic block's signals can be assigned to any signal names in the particular "instance".

Here is the list of signals and the instantiation section of the silly names version:

```
    // Inputs
    reg harry;
    reg bill;

    // Outputs
    wire andmary;
    wire ormaude;

    // Instantiate the Unit Under Test (UUT)
    and_or uut (
         .x1(harry),
         .x2(bill),
         .and2(andmary),
         .or2(ormaude)
    );
```

This testbench works just as well as the other. The simulation results in Fig. E.4 are the same except for the silly names.[4]



```
[2023-05-05 17:15:44 EDT] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
VCD info: dumpfile dump.vcd opened for output.
harry, bill = 0, 0,      andmary = 0,    ormaude = 0
harry, bill = 0, 1,      andmary = 0,    ormaude = 1
harry, bill = 1, 0,      andmary = 0,    ormaude = 1
harry, bill = 1, 1,      andmary = 1,    ormaude = 1
Finding VCD file...
./dump.vcd
[2023-05-05 17:15:44 EDT] Opening EPWave...
Done
```

Figure E.4: Simulation results: same with silly-name "instantiation".

Display of results in table form: The text display in the EDAPlayground "Log" window occurs because of the following lines of code:

```
     // display variables as text
     initial
          $monitor("x1, x2 = %b,%b, \t and2 = %b, \t or2 = %b", x1, x2, and2, or2);
     // show any changes in listed variables, in binary form
```

`initial` is a keyword that says, "Do this once." `$monitor` displays the listed signals any time one of them changes. The `%b` is a "modulus" indicator: it says "use binary number format." `\t` inserts a tab, just to make the table easier to read. If the timing-diagram simulation result is clear to you, skip the tabular form.

Input "stimulus" values and delay times: At last we reach the core of the simulation, where we stimulate the design with specified levels of the input variables:

```
     initial begin
          // Initialize Inputs
          x1 = 0;
          x2 = 0;

          // Wait 100 ns for global reset to finish
//        #100; // we won't do this, because it makes the display clumsy
```

[4]EPWave orders the signals alphabetically so the image does not visually match the one in Fig. E.2 but the result is the same. Simulation available at https://LAoE.link/Silly_Name_Simulation.html

```
          // Add stimulus here
          #10 x2 = 0; x1 = 1; // after a delay of 10 time units, apply new input levels
          #10 x2 = 1; x1 = 0;
          #10 x2 = 1; x1 = 1;
          #10 $finish;
     end
```

The `#10...` values say "wait ten time units before applying the input levels that follow." The testbench determines what a time unit is (usually nanoseconds, but for present purposes is are of no importance since we are only looking at the results of the design without concern for timing). So the set of stimuli shown above provide changes at 10ns intervals. This timing shows on the waveform displays such as Fig. E.4. Again, the keyword "initial" says, "Go through this sequence once" at startup.

E.2 Testbenches for sequential logic

Here is a testbench for a simple D flip-flop. We have created a task (similar to a function in C) named "display" to avoid repeating the code that writes the values of the d and q signals to the log window repeatedly.[5]

```
// D Flip-Flop Testbench
// manual clock
module test;

  reg clk;
  reg d;
  wire q;

  // Instantiate design under test
  d_flip_flop DFF(.clk(clk), .d(d), .q(q));

  initial begin
    // Dump waves
    $dumpfile("dump.vcd");
    $dumpvars(1);

    $display("Initialize signals");
    $display("Before first rising clock edge; q is unknown");
    clk = 0;
    d = 1;
    display;

    #5
    $display("Toggle clk; q should go to d input");
    clk = 1;
```

[5]Simulation available at https://LAoE.link/D_Flip-Flop_Simulation.html

```
    display;

    #2
    $display("Set d low; q should not change");
    d = 0;
    display;

    #2
    clk = 0;

    #5
    $display("Toggle clk; q should go to d input");
    clk = 1;
    display;

    #2
    #1  $display("Toggle d without a clock; q should not change");
    d = 1;
    #1
    d = 0;
    #1
    d = 1;
    $finish;
  end

  task display;
    #1 $display("d:%0h, q:%0h", d, q);
  endtask

endmodule
```

A testbench for sequential logic requires a clock. Here we manually toggle the clock but this becomes tedious for more complex logic, for example, a multi-bit counter. We can create up an automatic clock with an always block:[6]

```
// D Flip-Flop Testbench
//auto clock
module test;

  reg clk;
  reg d;
  wire q;

  // Instantiate design under test
  d_flip_flop DFF(.clk(clk), .d(d), .q(q));

  initial begin
    // Dump waves
    $dumpfile("dump.vcd");
    $dumpvars(1);
```

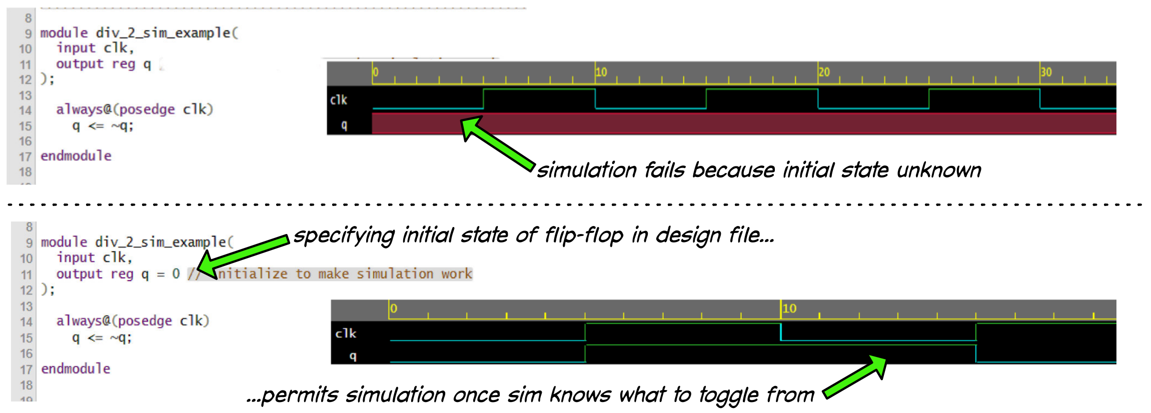
[6]Simulation available at https://LAoE.link/Auto_Clock_Simulation.html

```
    $display("Initialize signals");
    $display("Before first rising clock edge; q is unknown");
    clk = 0;
    d = 1;
    display;

    #5
    $display("Toggle clk; q should go to d input");
    display;

    #1
    $display("Set d low; q should not change");
    d = 0;
    display;


    $display("Toggle clk; q should go to d input");
    #7
    display;

    #3  $display("Toggle d without a clock; q should not change");
    d = 1;
    #1
    d = 0;
    #1
    d = 1;
    $finish;
  end

    // set up continuous clock
  always #5 clk = ~clk;

  task display;
    #1 $display("d:%0h, q:%0h", d, q);
  endtask

endmodule
```

E.2.1 Simulation of a sequential circuit must begin in a known state

Simulation can fail if your testbench fails to specify the circuit's initial state. In a combinational circuit, that will pose no problem, but it is a point that requires attention in a *sequential* circuit (a topic treated at §16N.1 and after). If, for example, you test a divide-by-two circuit and fail to tell the simulator what is the circuit's initial state, the simulator will balk, giving you no results.

You can specify the initial state in the design file. . . In Fig. E.5, a toggling flop circuit fails to simulate until we add a specification of the initial level of the flop output, in the design file (not in the testbench).[7]



```
module div_2_sim_example(
  input clk,
  output reg q
);

  always@(posedge clk)
    q <= ~q;

endmodule
```

```
module div_2_sim_example(
  input clk,
  output reg q = 0 //initialize to make simulation work
);

  always@(posedge clk)
    q <= ~q;

endmodule
```

0
10
20
30
clk
q
simulation fails because initial state unknown
specifying initial state of flip-flop in design file...
...permits simulation once sim knows what to toggle from

Figure E.5: The simulator needs to know the initial state of sequential elements.

. . . but instead one should provide a hardware initialization: The previous method makes the simulation work, but has no effect on the implementation – the startup condition is unpredictable. So a hardware reset should be added to the design:

```
module div_2_sim_example_better(
    input clk,
    input reset_bar,
    output reg Q
    );
   always@(posedge clk or negedge reset_bar)
   if (!reset_bar)
     Q <= 0;
   else
     Q <= ~Q;
endmodule
```

This hardware reset would take effect not only in simulation, but in the behavior of the programmed part.

E.3 Self-checking testbench

It is possible, though laborious, to write a testbench that predicts results for each function, and compares these predictions against actual results of a design. We did this for the programmable logic that linked the old 8051 micrcontroller to its peripherals in the "big board" version of the microcomputer labs used previously in this course. Here is an excerpt from the self-checking portion of the testbench: a truth table, and a "task" or function that refers to it.

[7]Simulation at https://LAoE.link/Initilization_Simulation.html

```
// Truth tables list expected outputs (rightmost column), and invoke
//  particular functions (named ``Task") to compare this prediction against
//  the result of the logic in the source file

  // here's the  truth table for CTR_OE_bar: f(br, loader)
     check_CTR_OE_bar(0,0,1);
     check_CTR_OE_bar(0,1,0);
     check_CTR_OE_bar(1,0,1);
     check_CTR_OE_bar(1,1,1);

// ...and the collection of "tasks" or functions invoked by truth tables

task check_CTR_OE_bar;
     input i_BR_bar;
     input i_LOADER_bar;
     input expect_CTR_OE_bar;

 begin
   #25; BR_bar = i_BR_bar; LOADER_bar = i_LOADER_bar;
   #25;
   if (CTR_OE_bar !== expect_CTR_OE_bar)
    begin
     $display("\t CTR_OE_bar ERROR: at time=%dns \t INPUTS: BR_bar=%b, LOADER_bar=%b,
     \t OUTPUT: CTR_OE_bar=%b, \t expected=%b", $time, BR_bar,LOADER_bar,
     CTR_OE_bar, expect_CTR_OE_bar);

     errors = errors + 1;
   end
 end
endtask
```

We will not try to explain this scheme fully, but will settle for the following sketch: the *truth table* for `CTR_OE_bar` shows two input levels and an expected output. The *task* named `check_CTR_OE_bar` watches for a mismatch between the truth table's output prediction and what the function `CTR_OE_bar` actually produces.

Any mismatch – where the actual and expected values don't match:

```
CTR_OE_bar !== expect_CTR_OE_bar
```

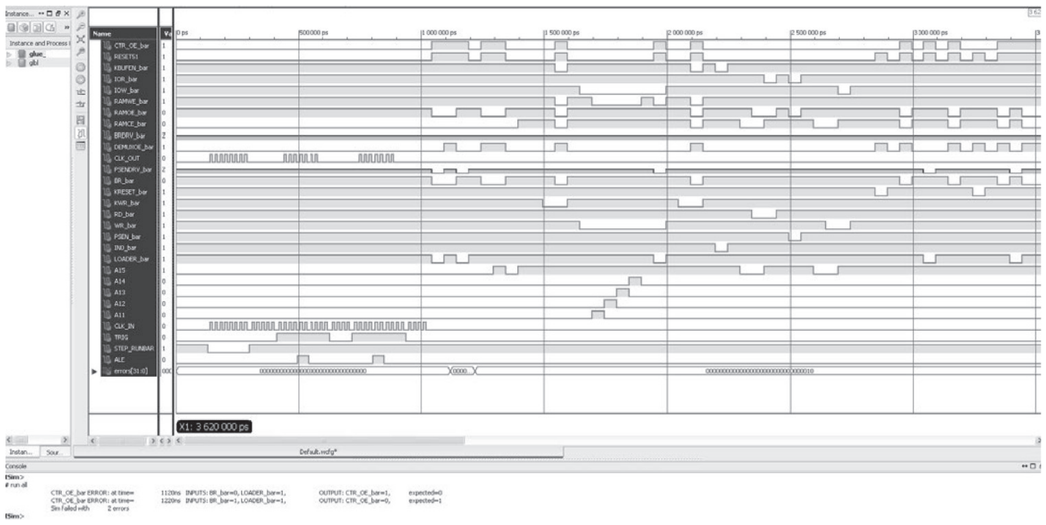
evokes a display detailing the mismatch: what was expected and what was observed:

```
 if (CTR_OE_bar !== expect_CTR_OE_bar)
       begin
            $display(
```

This self-checking feature is useful where you know what you expect and you want a warning that your design does not work without having to work through the timing diagram or textual output of the simulator.

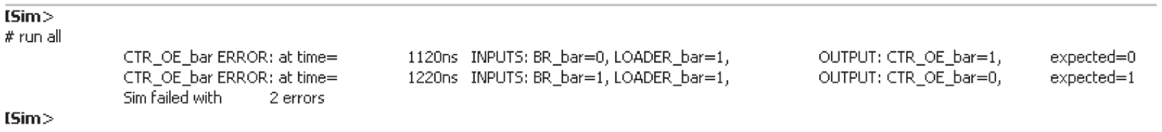
Without error-flagging, a complex testbench is hard to use: For example, if you were designing a processor with millions of gates you would expect to go through a lot of iterations to get it right and self-checking gives you an automatic way of seeing if your latest design works and where the problem is if it doesn't. Fig. E.6 is the result of simulating the much simpler gluePAL logic, but still a set of waveforms that one would be hard-pressed to use for spotting logical errors.

Because this testbench includes the self-checking feature, it lists the one function that failed to match expectations. The instance is listed at the bottom of Fig. E.6 – too small to read. Fig. E.7 has it in more readable form.



```
[Sim>
# run all
          CTR_OE_bar ERROR: at time=      1120ns  INPUTS: BR_bar=0, LOADER_bar=1,      OUTPUT: CTR_OE_bar=1,     expected=0
          CTR_OE_bar ERROR: at time=      1220ns  INPUTS: BR_bar=1, LOADER_bar=1,      OUTPUT: CTR_OE_bar=0,     expected=1
          Sim failed with        2 errors
[Sim>
```

Figure E.6: Testbench waveforms alone can make spotting logical errors very difficult.



```
[Sim>
# run all
          CTR_OE_bar ERROR: at time=      1120ns  INPUTS: BR_bar=0, LOADER_bar=1,      OUTPUT: CTR_OE_bar=1,     expected=0
          CTR_OE_bar ERROR: at time=      1220ns  INPUTS: BR_bar=1, LOADER_bar=1,      OUTPUT: CTR_OE_bar=0,     expected=1
          Sim failed with        2 errors
[Sim>
```

Figure E.7: Self-checking testbench picks out the single function that failed from the simulation plotted in Fig. E.6.

Please send comments or corrections to: authors@LAoE.link