Verification

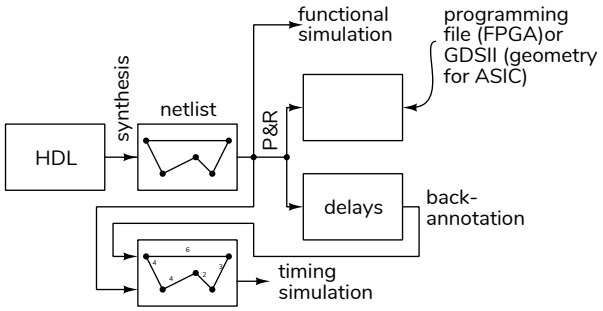
This lecture describes how digital systems are tested.
After this lecture you should be able to select an appropriate verification strategy including: selecting simulation or hardware testing; stimulus-only or self-checking testbenches; selection of test inputs; use of "known-good" models; unit testing; regression testing; distinguish between functional (RTL) and gate-level (timing) simulations; use delays and event controls to generate waveforms in a System Verilog testbench.

Design Verification

Verification is checking that a design meets requirements. Verification typically requires more time and effort than the initial design and is typically done by simulation.

Functional (RTL) versus Timing (Gate-Level) Simulation

The following diagram shows the steps involved in the design and verification of a digital logic circuit.



The logic synthesizer generates a netlist that describes how the components (gates, flip-flops, etc) are connected. The place and route (P&R) step places the components at specific locations on the IC and connects them using the metal layers of an ASIC or a PLD's routing resources. The P&R step determines the delays between components.

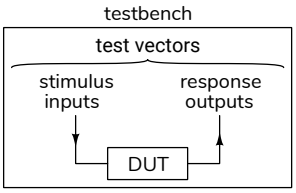
Simulations can verify both the correct functioning of the design ("functional verification") and that it will operate at the required clock frequency ("timing simulation").

Functional testing verifies the design by assuming zero propagation delay through combinational logic and interconnects. This checks that the logical ("functional") design is correct. This can be done before mapping the design to gates and placing these at specific locations of the IC because propagation delays do not affect the results.

Timing simulation verifies that the design will behave correctly with the actual signal delays that will appear in the final design. This requires that the delays estimated from P&R are added ("back-annotated") to the netlist.

Types of Testbenches

A simulation consists of the device (or design or unit) under test (DUT/UUT), plus additional code called a testbench that applies inputs to the DUT and checks its output:



Stimulus-Only

The simplest testbench applies inputs to the DUT and saves the inputs and outputs to a file so a designer can view them. These testbenches are useful during the initial design process.

Self-Checking

Once initial testing is complete, it's desirable to ensure that subsequent changes to the design do not introduce errors ("regressions"). Manually checking the outputs after each change is tedious and error-prone. Once the expected outputs have been established, a "self-checking" testbench can check the outputs itself and flag any differences.

A self-checking testbench sets the inputs and waits for a fixed delay or for an event indicating the DUT

output is valid. The code then compares the DUT output to the desired result.

Generating Test Vectors

Test vectors are the values to be applied to the DUT and the corresponding expected outputs.

Test vectors can be generated by the testbench itself (e.g. in a loop or using a random number generator) or they can be read from a file generated by other software.

Inputs

Usually there are too many possible combinations of inputs to be able to test them all. However, enough test vectors should be generated to ensure a reasonable confidence in the correct operation of the design.

Test vectors should include:

1. typical input values,
2. minimum/maximum valid input values,
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) an 4x8-bit multiplier with a 10-bit output, (b) a frequency divider using an 8-bit timer?

Outputs

For very simple designs it may be possible to compute the correct outputs manually. But for more complex designs this would take too long or be too error-prone. In this case the correct outputs have to be generated by software.

This requires that there be a "known-good" software model of the desired behaviour that has been *independently* verified. How this is done depends on the application.

Test Strategies

Unit Testing

It's often more effective to test components of a design individually rather than the complete design. This "unit testing" makes it easier to isolate the source of a problem.

It's often useful to start testing before a design is complete. As each part of a design is completed, testbenches, tests vectors and scripts are prepared and added to the test suite for regression testing.

Test Automation

Running these tests manually would take too long and be too error-prone. Scripts are used to automate testing by compiling the code, running simulations and summarizing the results.

Many EDA (Electronic Design Automation) tools, including the FPGA design and test software from Intel and Xilinx can be controlled by scripts written in a simple scripting language called tcl (Tool Command Language, pronounced "tickle"). For example, the various programs in the Quartus tool suite have embedded tcl interpreters and many of the configuration files are actually tcl scripts that set variables.

tcl is a very simple scripting language. Strings are the only data type. The first word of each line is the command to be execute. Commands within brackets ([]) are executed and the resulting string is substituted in place of that command.

Exercise 2: What two tcl commands are executed by the following tcl script: `set x [expr 1 + 1]`?

Verilog for Verification

Early integrated circuits were designed and laid out by hand. As complexity increased it became necessary to simulate these circuits before they were manufactured to be reasonably sure that they would work properly. The Verilog language (from "verification" and "logic") was designed to simplify the simulation of these digital circuits.

It later turned out that [a subset of] a language designed to model hardware for simulation purposes was also well-suited as the input language to a logic synthesizer.

In this section we will cover a few additional features of Verilog that are useful for simulation.

Parallel Statements

A module can contain any number of **`initial`** and **`always`** blocks that execute concurrently:

`initial` blocks execute once at the start of the simulation. Most Verilog testbenches run through

their test vectors sequentially using one or more **`initial`** block(s).

`always` blocks execute continuously.

Multiple statements may appear in each block between **`begin`** and **`end`**.

Delays

Simulations need to model propagation delays and clock periods.

When a delay is specified, the default time unit is picoseconds. However, this can be changed with the **`` `timescale``** directive which defines the default units and the resolution. For example:

```
`timescale 1us/1ns
```

means delays are given in microseconds and delays of less than 1 ns cannot be resolved. The suffixes **`ps`**, **`ns`**, **`us`** can be used for pico-, nano- and micro-seconds.

Note that these delays are not synthesizable because modern logic design does not use delay circuits.

Delays and Event Control

The execution of the next statement is often delayed, in simulation time, by one of the following:

#number delays execution by *number* simulation time. (**`#0`**) delays until after un-delayed events at the same time. A (**`#1step`**) delay pauses execution by the smallest possible amount – the time resolution.

@(*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 the signal value. Multiple events can be given separated by **`or`** (e.g. **`@(posedge clk or reset)`**).

We have used event controls to control execution of **`always_ff`** procedural blocks but they can also be used in simulations to synchronize execution of procedural blocks (e.g. **`@(count == 0)`**).

wait(*expression*) delays execution until the expression is non-zero.

Exercise 3: What's the difference between `wait(x) y='1;` and `@(x) y='1;`?

Sequential Statements

These statements appear within **`always`** or **`initial`** procedural blocks and execute sequentially (one after the other). They useful when writing testbenches for simulation (they can also be used for synthesis – but not in this course).

`for`, **`while`** and **`do`** loops are the same as in C. The **`break`** and **`continue`** statements from C can also be used.

The **`if`**/**`else`** statement syntax is the same as in to C.

The **`begin`** and **`end`** keywords group statements that should be executed together. They serve the same purpose as braces ({, }) in C (which are used for concatenation and replication in Verilog).

System Tasks

Functions with names beginning with **`$`** are system tasks. Some examples: **`$display()`** and **`$warning()`**, similar to C's **`printf()`**, can be used to print messages during simulation; **`$finish`** and **`$stop`** terminate or suspend a simulation. **`$dumpfile`** and **`$dumpvars`** record changes in signals to a "value change dump" (**`.vcd`**) file for subsequent viewing with a waveform viewer.

Example

The following demonstrates how the Verilog features described above can be used in writing a testbench. The testbench is often in a separate file. In this case the testbench is hidden from synthesizers by **`synthesis translate_on/off`** pragmas.

```
`timescale 1us/1ns

// the DUT (device under test): a slightly buggy
// synchronous adder (for which 1+1!=2)

module sadder
  ( input  logic [3:0] a, b,
    output logic [3:0] c,
    input logic reset, clk ) ;

   always_ff @(posedge clk)
     c <= reset || a==4'd1 && b==4'd1 ? '0 : a + b ;

endmodule

// synthesis translate_off

// // the testbench

module ex59 ;
```

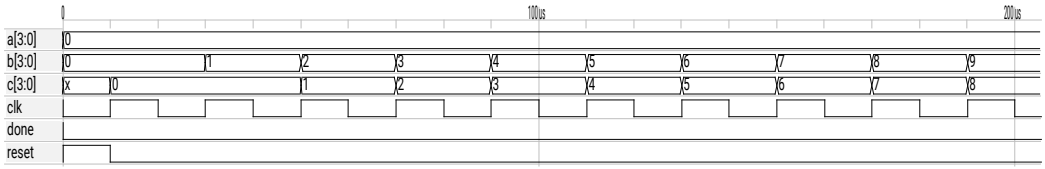


Figure 1: Testbench simulation results.

```
   // DUT inputs & outputs
   logic [3:0] a, b, c ;
   logic reset, clk ;

   // set their initial values
   initial begin
      {a,b,clk} = '0 ;
      reset = '1 ;
   end

   // instantiate the DUT
   sadder s0 (.*) ;

   // simulation-complete flag
   logic done = '0 ;

   // continuous 20ns clock
   always #10 clk = ~clk ;

   // release reset immediately after first rising
   // edge
   initial @(posedge clk)
     #0 reset = '0 ;

   // run through test vectors
   initial begin

      // create a VCD waveform file
      $dumpfile("ex59.vcd") ;
      $dumpvars ;

      // check all possible inputs
      for ( integer i=0 ; i <= 15 ; i++ ) begin
         for ( integer j=0 ; j <= 15 ; j++ ) begin

            // wait for rising clock edge plus one
            // simulation time step
            @(posedge clk) #1step ;

            // check the output
            if ( reset && c != '0 || c != a+b )
               $warning
                 ("error:a=%b,b=%b,c=%b\n",a,b,c) ;

            // set the next inputs
            {a,b} = {i[3:0],j[3:0]} ;

         end
      end

      done = '1 ;
   end

   // end simulation when done
   initial
     wait (done) $finish ;

endmodule

// synthesis translate_on
```

Running this testbench prints:

```
# ** Warning: error:a=0001,b=0001,c=0000
#    Time: 370001 ns  Scope: ex59 File: ex59.sv Line: 63
# ** Note: $finish    : ex59.sv(77)
#    Time: 5110001 ns  Iteration: 1  Instance: /ex59
# End time: 19:56:52 on Mar 23,2022, Elapsed time: 0:00:00
# Errors: 0, Warnings: 2
```

and the initial portion of the waveforms dumped to the file **`ex59.vcd`** are shown in Figure 1.

Simulation vs Hardware Testing

Programmable logic (FPGA and CPLD) designers have the option of testing a circuit in addition to simulating their designs.

Simulations have several advantages:

- simulators give more visibility into the operation of the design than hardware (even when using embedded logic analyzers such as SignalTap),
- compilation is much faster than synthesis,
- simulations can be automated (e.g. to run nightly regression tests),
- it's relatively easy to supply test data ("test vectors") to an FPGA being simulated and collect and process the results,

on the other hand:

- a simulation is orders of magnitude slower than hardware, and so:
- a simulator cannot process input in real time,
- a simulation cannot model all details of the final design (interfaces, power supplies, etc.).

Thus simulations tend to be used early in the design process followed by testing on the final hardware configuration.