ELEX 7660 : Digital System Design
2018 Winter Term

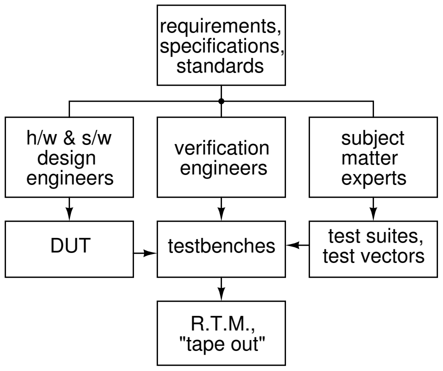
Verification

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; use of BFMs; unit testing; regression testing. vectors and test benches; distinguish between functional (RTL) and gate-level (timing) simulations; use delays and event controls to generate waveforms.

Design Verification

Verification means testing a design to verify that it meets requirements. The effort required to verify a design often exceeds that required for the design.

Verification of larger designs involves several groups:


requirements, specifications, standards
h/w & s/w design engineers
verification engineers
subject matter experts
DUT
testbenches
test suites, test vectors
R.T.M., "tape out"

To "re-spin" an ASIC that has a errors that render it unusable typically requires months and hundreds of thousands (or millions) of dollars. Such errors are often "fatal" to a project because the product misses a market window or for financial reasons. ASIC designers must therefore simulate their designs carefully before "taping out" to manufacturing.

On the other hand, recompiling an FPGA design to fix an error may only take hours and in many products a new FPGA configuration can be included in a firmware update. This allows the FPGA design to be updated even after the product is in the field. Thus the risk posed by FPGA design errors is much lower and this affects the choice of verification strategies.

Simulation vs Hardware Testing

FPGA designers have the option of testing the FPGA hardware as well as 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 easily 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,

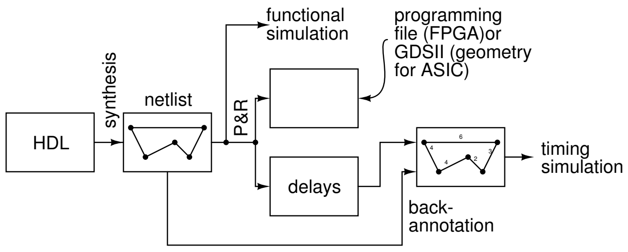
on the other hand:

- 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.

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 then using FPGA routing resources or the metal layers of an ASIC. The P&R step determines the delays between components.

Simulations can be used to verify the functionality of a design and also that it will operate at the required clock frequency.

Functional testing verifies the design by assuming zero propagation delay through combinational logic. This checks that the logical or functional design is correct. This can be done before the design is mapped into gates and placed on specific portions of the chip 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 "back-annotated" to the netlist.

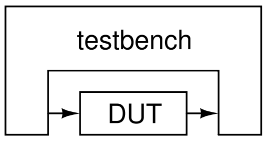
Synthesis software uses models of the delays through the device's logic blocks and interconnects along with timing constraints for external devices (to be covered later) to guide the P&R process to try to ensure that timing requirements will be met.

FPGA designs typically rely on "static" (no simulation) timing analysis (STA) to verify that the various timing constraints will be met. However, timing simulations for FGPA designs can be used to check that the designer has not missed any timing constraints.

However, ASIC designs will typically run extensive timing simulations to ensure the device will operate properly before "taping out" and sending the device to be fabricated.

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 simply applies inputs to the DUT and dumps the inputs and outputs to a file so the they can be viewed by the designer. These are mainly useful during the initial design process.

Self-Checking

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

Generating Test Vectors

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

The test vectors can be generated by the testbench itself (e.g. in a loop or 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 all of them. 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.

For example, when testing a tone generator designed for frequencies between 300 Hz and 3 kHz we might test it at frequencies of 1 kHz (typical), 300, 3 kHz (limits) and 0 Hz (invalid) and a range of random frequencies.

Exercise 1: What test inputs might be appropriate for testing a 16×8-bit multiplier?

Coverage testing is another approach that can be used to ensure an adequate number of test vectors. A simulator can monitor the parts of the DUT that are executed during a simulation and highlight the parts of the design that were not used during the simulation. Test vectors can then be added to ensure more complete test coverage.

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.

For example, when testing a CPU there might be application test suites of programs and the results of running them. For a signal processor there could be Matlab scripts that are known to produce the correct output waveforms. Some standards include tests that can be used to check for conformance to the standard. And if hardware already exists (e.g. an existing version of the design) it can sometimes be used to generate test vectors (allowing the new design to replicate errors or undocumented features in the original design).

Test Strategies

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.

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 is also well-suited as the input language to a logic synthesizer.

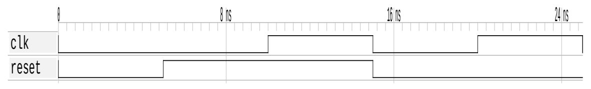
In this section we will cover a few additional features of Verilog that are useful for simulation.

Initial Blocks

Initial blocks can generate clocks and reset signals as in the following example:

```
'timescale 1ns/1ns

module ex20 ;

  logic reset, clk ;

  initial begin
    reset = '0;          #5
    clk = '0;
    #5ns reset = 1;
    #5ns clk = 1;
    #5ns reset = 0;
    clk = 0;

    forever
      #5ns clk = ~clk;
  end

  initial
    #30 $finish ;

endmodule
```

generating the following clock and reset signals:



Most Verilog testbenches run through their test vectors sequentially using an `initial` process.

For simple DUTs each test vectors sets the inputs and the code waits for an event indicating the DUT output is valid or for a fixed delay. The code then compares the DUT output to the desired result.

For more complex DUT's the testbench needs to apply test vectors through "transactions" on a bus functional model (BFM). These buses can be processor (e.g. NiosII, ARM) or peripheral (e.g. SPI, USB) buses. For example, a testbench may generate memory read/write cycles on a processor bus or generate and respond to a sequence of message on a peripheral bus according to a particular protocol. In most cases simulation IP is available for these buses.

Simulation Time Control

Delays

Delays are not synthesizable. They are used to model the behaviour of devices (e.g. propagation delays through gates) or to create waveforms in testbenches. In this course we only cover the latter.

The syntax #*n* before a sequential statement suspends execution of that block for simulation time *n*.

However, this can be changed with the `'timescale` directive which takes two values: the default units and the resolution as shown in the example above. The default units are used if no unit is specified in a delay. Resolution specifies the quantization of time events.

Event Control

The event control expression @(*event*) before a statement pauses execution until *event*. The event can be `posedge` or `negedge` before a signal name or just the signal name. The latter refers to any change in the signal value. Multiple events can be given separated by `or`.

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.

`wait()`

The `wait()` control pauses execution of the associated statement until the specified condition is true.

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

Delay in RHS of Assignments

Putting a delay or event on the right-hand side (RHS) of the assignment causes the RHS to be evaluated immediately and an update scheduled after the specified delay or event. This can be used in a non-blocking assignment to schedule a future change to a signal.

Input/Output

For printing messages during simulations, the `$display()` (or `$write()`) system tasks can be used.

In addition to the `$readmemh()` system task that reads a complete data structure directly into memory, System Verilog supports C-style file I/O using similarly-named system tasks such as `$fopen()`, `$fread()`, etc. These can read and write text or binary files (or to/from strings). These are more practical when reading or writing large files. See that standard for details.

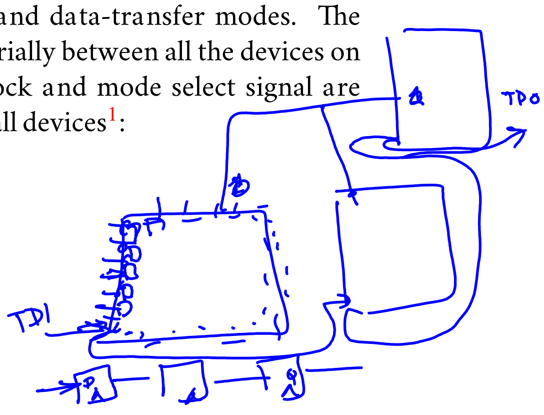
Boundary Scan and JTAG Interfaces

As the number of pins on IC packages grew and pin pitch decreased it became more difficult to test assembled printed circuit boards using "flying probe" or "bed of nails" testers.

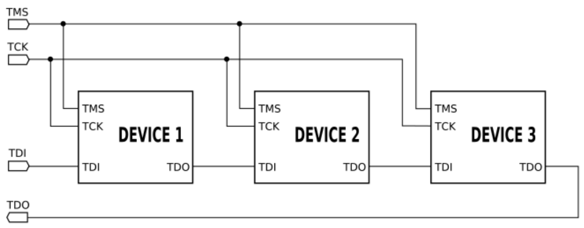
On the other hand, increased IC density made it possible to add test circuit to each IC pin that allows the pin to be disconnected from internal logic and set or read under external control.

"JTAG" (Joint Test Action Group) is a standard originally developed for boundary-scan testing of ICs. It is a clocked serial interface that passes data through a long chain of flip-flops (i.e. a shift register). Each flip-flop can thus set or read the state of one pin.

A JTAG port has serial data in (TDI) and out (TDO) pins, a clock signal (TCK) and a test-mode-select (TMS) signal that is used to switch between the interfaces's control- and data-transfer modes. The data pins pass data serially between all the devices on a board while the clock and mode select signal are applied in parallel to all devices[1]:



[1] From Wikipedia.



JTAG interfaces can also be used to program FPGAs, PLDs and various type of memory. They are also used to interface with on-chip debugging features such as logic analyzers, software debuggers, consoles and others.

Assertions

An assertion is a declaration that something is true. Assertions can be used to detect logical errors in Verilog simulations. Assertions added to code to check that conditions that the designer expects to be true are, in fact, true.

When an assertion fails the simulation stops and the location of the failed assertion is printed. The designer must then figure out the cause of the problem.

Assertions simplify finding bugs because the problem is isolated to a specific portion of the code.

Assertions are not synthesizable and are ignored by synthesizers.

Assertions can be included at the start and end of procedural blocks to check that all inputs and outputs are within expected ranges and consistent.

For example, if a module's x input is always expected to lie between 300 and 3000, we might include code such as:

```
always_comb begin
  assert( x >= 300 && x <= 3000 ) ;
end
```