"If you don’t test it, it isn’t going to work"
- Mark Horowitz
Testing

• Logic blocks (*.v)
  – Will be synthesized into hardware circuits
• Testing blocks (*.tb.v OR *.vt)
  – Pretty much anything goes
• Instantiate logic blocks inside testing blocks; drive inputs and check outputs there
• Examples of things that are ok in *.vt but not ok in *.v modules in this class unless you are told otherwise
  – “#” delay statements are essential in testing modules and should never be in hardware except D FFs for “clock to Q” delays
  – “signed” regs and wires are extremely useful for printing 2’s complement signals
  – “for” loops
  – any statements for which the resulting hardware is unclear
Example test module

- Basic flow Approach 1
  - All signals are generated by test code in the test module
  - Easier and quicker to set up
  - Cumbersome for very large numbers of test cases

```verilog
in    = 4'b0000;
clk   = 0;
reset = 1;
#100; clk=1; #100; clk=0;
reset = 0;
#100; clk=1; #100; clk=0;
in    = 4'b0001;
#100; clk=1; #100; clk=0;
in    = 4'b1010;
#100; clk=1; #100; clk=0;
...  
```
Example test module

- Basic flow Approach 2
  - Both “test” block and “hardware” block are coordinated by the same clock signal which is generated by an independent clock oscillator module in the test module
  - Better for more complex systems
  - Adds a small amount of realism in the timing of input and output signals
Example test module

• Basic flow Approach 2
  – A very simple clock oscillator could look like this:

    ```verbatim
    initial begin
      clock = 1'b0; // must initialize
      ...
      #10000;       // main simulation
      ...
      $finish;      // stop simulation
    end

    // osc inverts clock value every #100
    always begin
      #100;         // cycle time = #200
      clock = ~clock;
    end
    ```
  – A slightly better design would use a clock reset signal
Example test module

- Basic flow Approach 2
  - Here is an example of how code in the test generator could look:

```verilog
initial begin
    reset = 1'b1;
data   = 8'h00;
clock  = 1'b0; // must initialize #1000; // wait
    @(posedge clock); #10; // clk-to-Q
reset = 1'b0; // clear reset
    @(posedge clock); #10;
mag_data = 8'h0F;
    @(posedge clock); #10;
mag_data = 8'hB7;
    @(posedge clk); #10
    @(posedge clk); #10
    repeat (50) @(posedge clk); // wait
$finish; // stop simulation
end
```
Verification

• A number of ways to verify designs:
  1) Eyeball text printouts
     • Quickest and easiest
  2) Eyeball waveforms
     • Quick and easy for simple designs
  3) Write reference code and some other code to see if the two are the same. Make sure you temporarily force an error to test your setup.
     • This is the most robust and is what is required for non-trivial designs

• As designs become more complex, verifying their correctness becomes more difficult
Verifying Correctness

• “Golden reference” approach
  – Write an easy to understand simple model in a higher-level language
    • C or matlab commonly
      – Matlab is a natural choice for DSP applications
    • Must be written in a different way from the verilog implementation to avoid repeating the same bugs
  – Designers agree the golden reference is the correct function (imagine your colleagues participating in a detailed design review)
  – Many high-level tests must be run on golden reference
    • Model should be fast
    • Imagine days of simulations on tens or hundreds of computers
Golden Reference Approach

• There are two major approaches to comparing with the Golden Reference:

1) Hardware and Reference must be “Bit-accurate” or “Bit-perfect” or “Bit-true”
   • Hardware must match golden reference exactly, bit for bit, cycle by cycle
   • Far easier to automate testing
   • Possibly less testing will be needed than approach #1
   • Golden Reference must now do awkward operations such as rounding and saturation that exactly match hardware
     o For example, floor(in + 0.5) for rounding

2) Hardware and Reference must be "close"
   • Possibly compare SNR of hardware vs. reference
   • Inadequate for control hardware which must be a perfect match
   • Possibly more testing will be needed than approach #2
Golden Reference Approach

• Implementing the “checker” comparison tool
  1) Bit-accurate: comparisons could be done in the verilog testbench, in matlab, in linux using the built-in “diff” command, or by using many other tools
  2) Close enough: Almost certainly it will be much easier to check in matlab than using any other tool
Sources of Input Test Data

• From a data file on disk
  – For example, from a video test sequence

• From data generated by a script
  – For example, all values 0-65,535 for a block with 16 binary inputs
  – Input data may be generated from either matlab or verilog. I think it's a little easier to generate input data in verilog and then print both input and output to a matlab-readable *.m file and test and compare in matlab
  – It can sometimes be a little awkward to read data from a file in verilog
  – You may find it handy to declare variables as signed in verilog and print them using $fwrite so both positive and negative numbers print correctly