

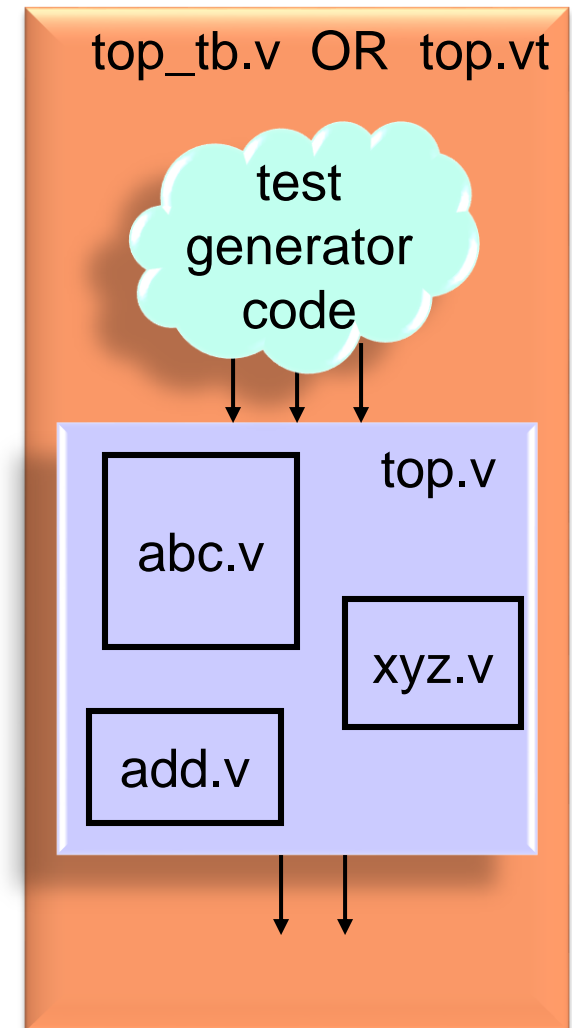
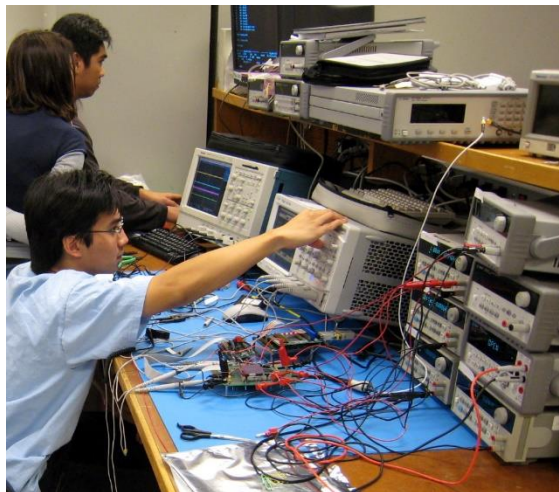
VERILOG 5: TESTING

“If you don’t test it, it isn’t going to work”

- Mark Horowitz

Testing

- Hardware blocks (*.v)
 - Will be synthesized into hardware circuits
- Testing blocks (*.vt OR *_tb.v OR tb_*.v)
 - Also called the “testbench”
 - Pretty much any code is ok
 - However it should always be clear
- Instantiate hardware inside the testbench; drive inputs and check outputs there



Verilog Code in Testbenches

- Examples of verilog code that are ok in testbenches but not ok in hardware modules in this class unless you are told otherwise

- “#” delay statements are essential in testing modules and should never be in hardware (except for “clock to Q” delays in D FFs)
- “signed” regs and wires are extremely useful for printing 2’s complement signals
- “integer” variables for “for loops”, and counters
- “for” loops
- `$write("format", var1, var2, ...)`

// writes text to screen or file using the specified

// format with optional variables. Example:

// `$write("in = %b, out1 = %b, ", in, out1);`

// `$write("out2 = %b, out3 = %b", out2, out3);`

// `$write("\n");`

`$display("format", var1, var2, ...)`

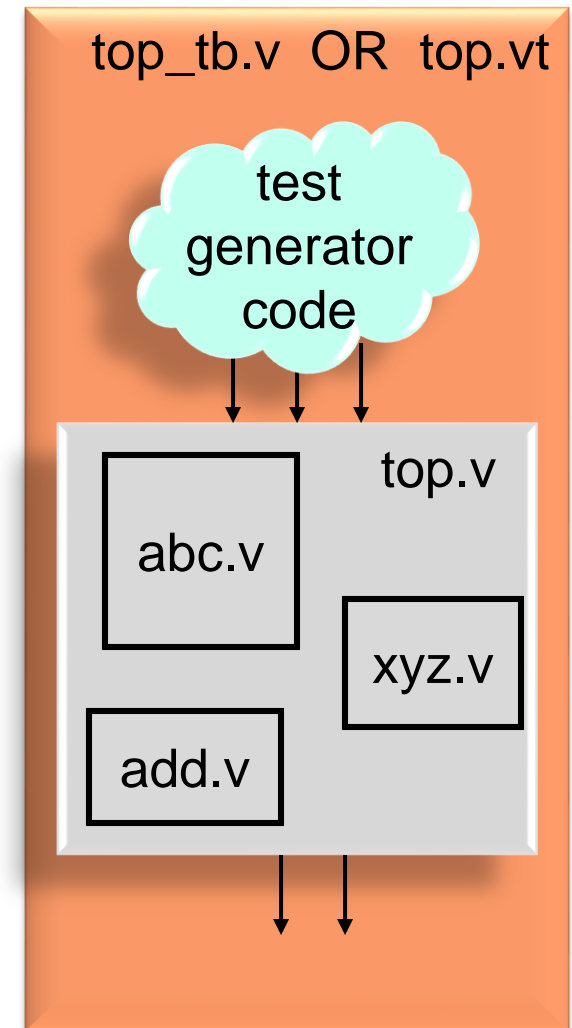
// same as `$write` except it adds a `\n` new line

// automatically. I generally prefer `$write` so I can add

// my `\n` exactly when I want it at the end of a series of

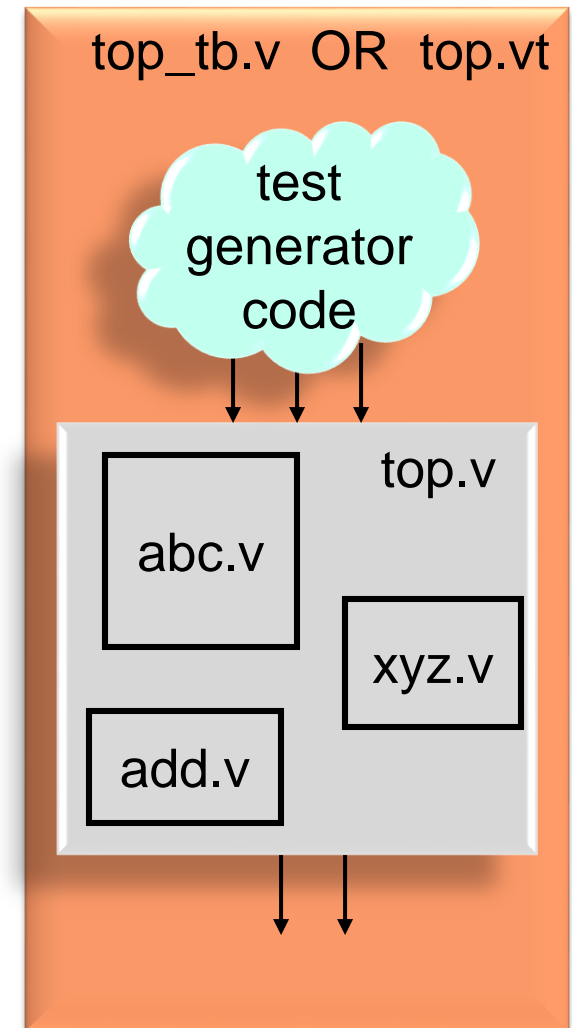
// `$write` statements.

See
Sutherland
handout,
page 38, for
details on
printing
format
specifiers



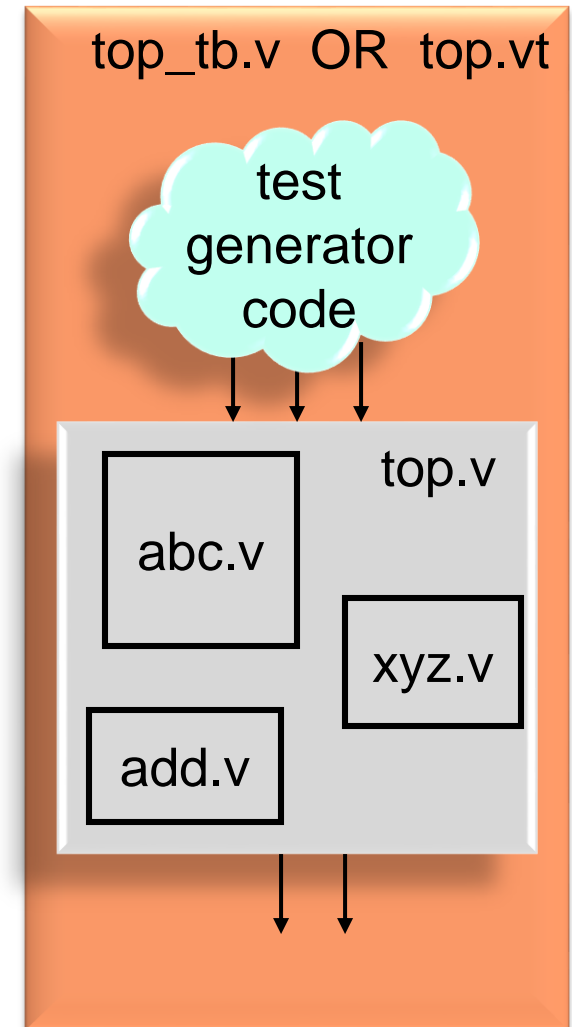
Verilog Code in Testbenches

- Examples of verilog code that are ok in testbenches but not ok in hardware modules in this class unless you are told otherwise
 - `@(posedge clock);` // only for testbenches when
`@(negedge clock);` // used as a standalone
// statement that waits for the
// next positive or negative
// edge of "clock" in this case
 - `repeat (50) @(posedge clk);`
// the "repeat" statement can
// be very handy for
// repeatedly executing a
// statement (or a block of
// statements)



Verilog Code in Testbenches

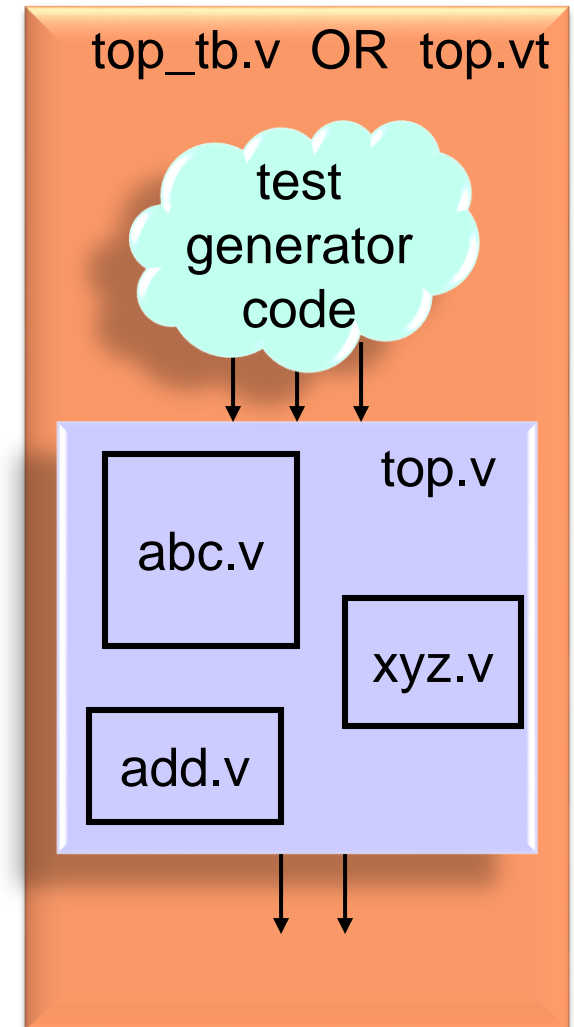
- Examples of verilog code that are ok in testbenches but not ok in hardware modules in this class unless you are told otherwise
 - `$stop;` // Halts the simulation.
 // This is probably the better one to use
 // for Modelsim because \$finish causes
 // Modelsim to ask if you really want to
 // quit the simulator which is probably
 // not what you want.
 - `$finish;` // Ends the simulation.
 // This is probably the better one to use
 // for Cadence and Synopsys verilog
 // simulators running by command line
 // on linux because \$stop causes the
 // simulator to drop back to an
 // interactive command line prompt
 // rather than the linux command line.



Verilog Code in Testbenches

- Examples of verilog code that may appear in either testbench modules or hardware modules

- ``timescale time_unit base / precision base`
 - // The first argument specifies “#1” delay. The
 - // second argument specifies the precision with
 - // which delays may be specified.
 - // Base values may be 1, 10, 100, or 1000; and its units
 - // may be {s,ms,us,ns,ps,fs}
 - // Ex: ``timescale 1ns/10ps`
- Cadence NC Verilog simulator
 - Requirement: ``timescale` must be at the top of the first file listed in the .vf file
 - Best case: place ``timescale` at the top of the .vt file, and list the .vt at the top of your .vf
 - Alternatively: place ``timescale` at the top of every .v and .vt file
- Modelsim simulator
 - Put a ``timescale` command as the first line in your top-level testbench file (e.g., `top_tb.v`). No need to put it in any of your hardware modules.



Testbench Design: Approach 1

Basic Flow

- All signals including the clock are generated by test code in the test module
- This approach is easier and quicker to set up compared to approach #2
- But it is cumbersome for testbenches requiring a very large numbers of test cases

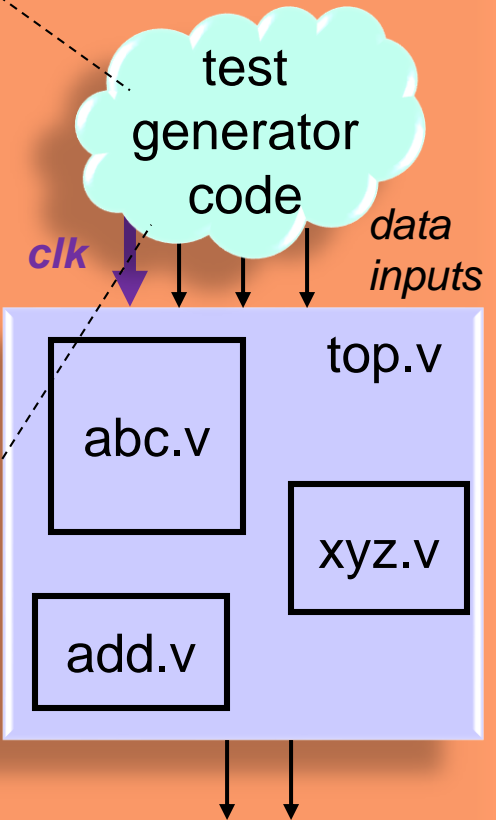
```
initial begin
  in    = 4'b0000;
  clk   = 1'b0;
  reset = 1'b1;
  #100; clk=1'b1; #100; clk=1'b0;

  reset = 0;
  #100; clk=1'b1; #100; clk=1'b0;

  in    = 4'b0001;
  #100; clk=1'b1; #100; clk=1'b0;

  in    = 4'b1010;
  #100; clk=1'b1; #100; clk=1'b0;
  ...
  $stop;
end
```

top_tb.v OR top.vt



Testbench Design: Approach 1

Basic Flow

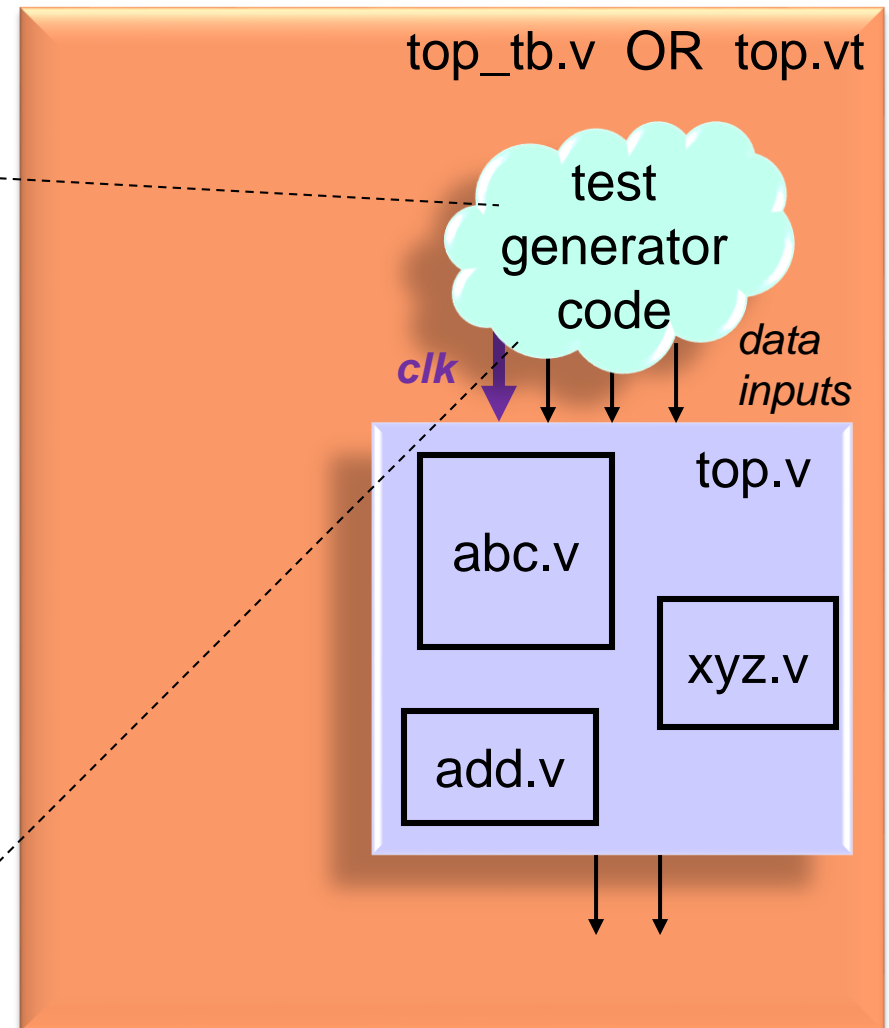
- An *Approach 1* example testbench including \$write()

```
initial begin
  in    = 4'b0000;
  clk   = 1'b0;
  reset = 1'b1;
  in2   = 8'hA5;
  #100; clk=1'b1; #100; clk=1'b0; // 1 clk
  $write("in = %b, out = %b\n", in, out);

  reset = 0;
  #100; clk=1'b1; #100; clk=1'b0; // 1 clk
  $write("in = %b, out = %b\n", in, out);

  in    = 4'b0001;
  #100; clk=1'b1; #100; clk=1'b0; // 1 clk
  $write("in = %b, out = %b\n", in, out);

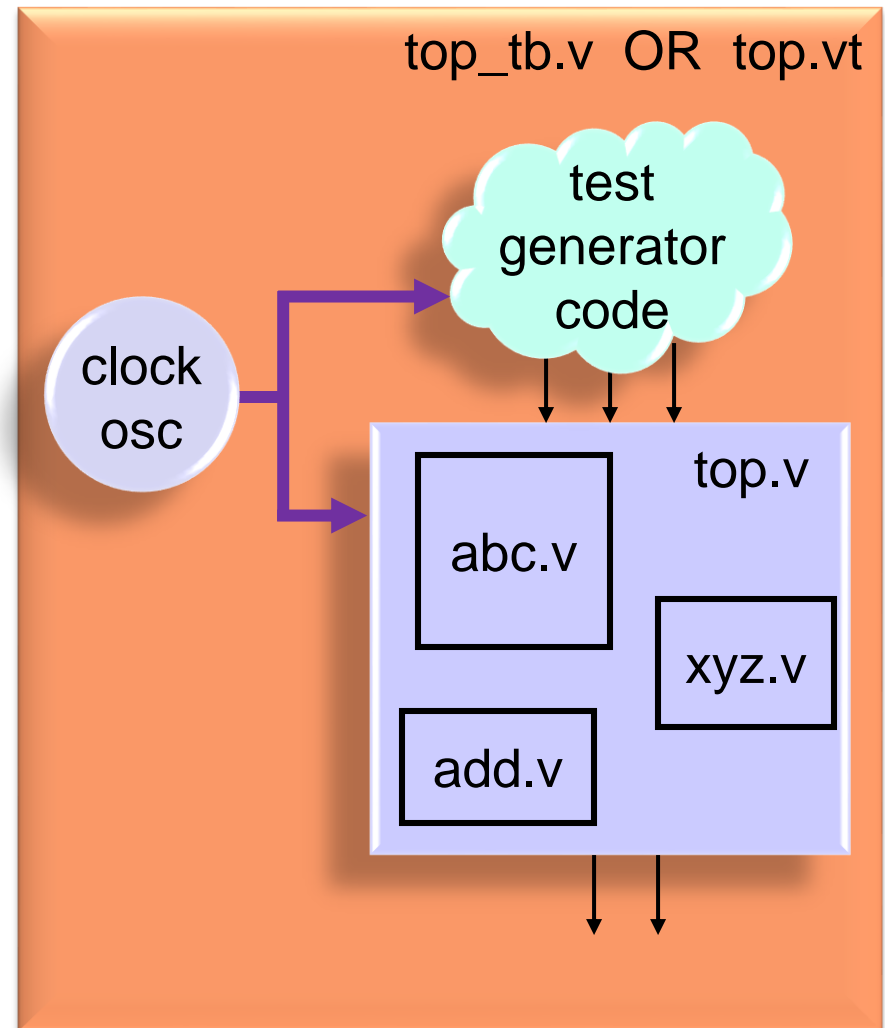
  in    = 4'b1010;
  #100; clk=1'b1; #100; clk=1'b0; // 1 clk
  $write("in = %b, out = %b\n", in, out);
  ...
  $stop;
end
```



Testbench Design: Approach 2

Basic Flow

- Both the “test” block and the “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 some realism in the timing of input and output signals



Testbench Design: Approach 2

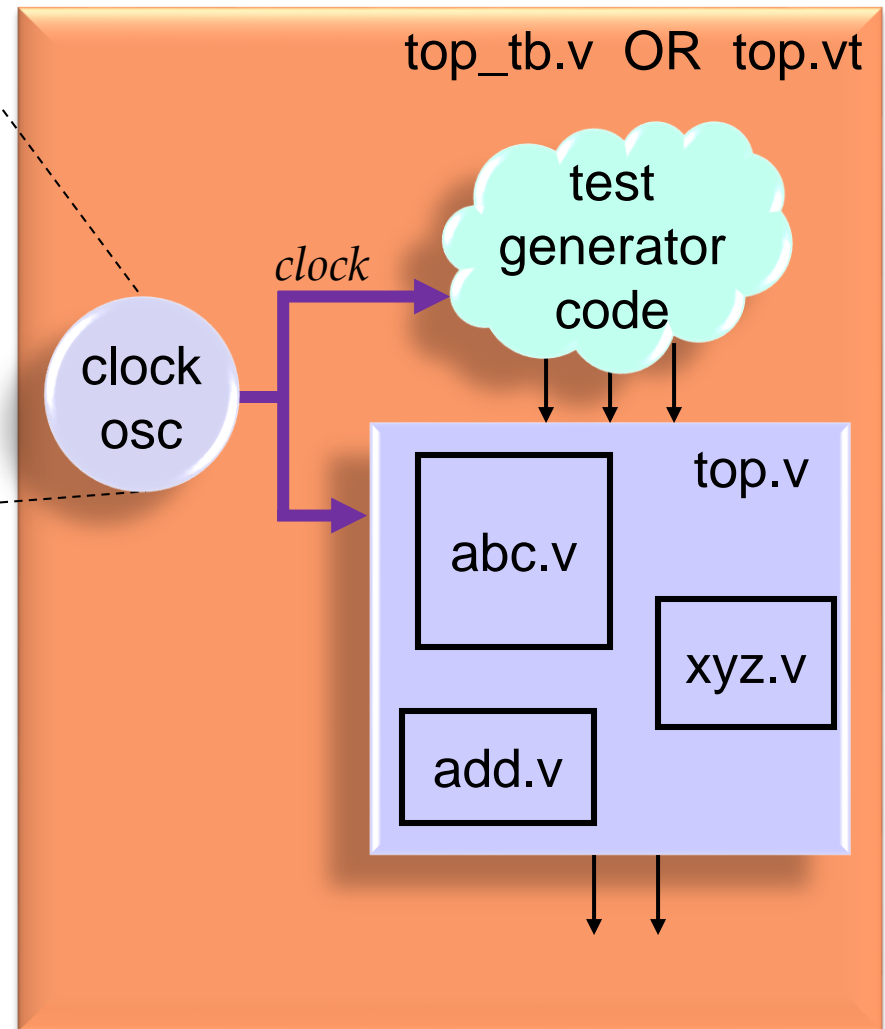
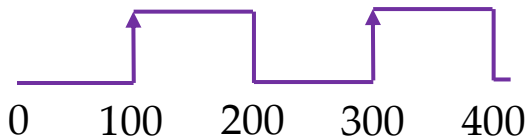
Example Clock Oscillator

```
reg clock;

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

// osc inverts clock value every #100
always begin
    #100;            // cycle time = #200
    clock = ~clock;  // invert clock
end
```

- This code is error prone because two different blocks set the reg *clock*. To avoid a problem, the **initial** block sets *clock* at time=0 and the second block waits until time=100 and later to set *clock*
- A better design would use a *reset* signal to initialize *clock*



Testbench Design: Approach 2

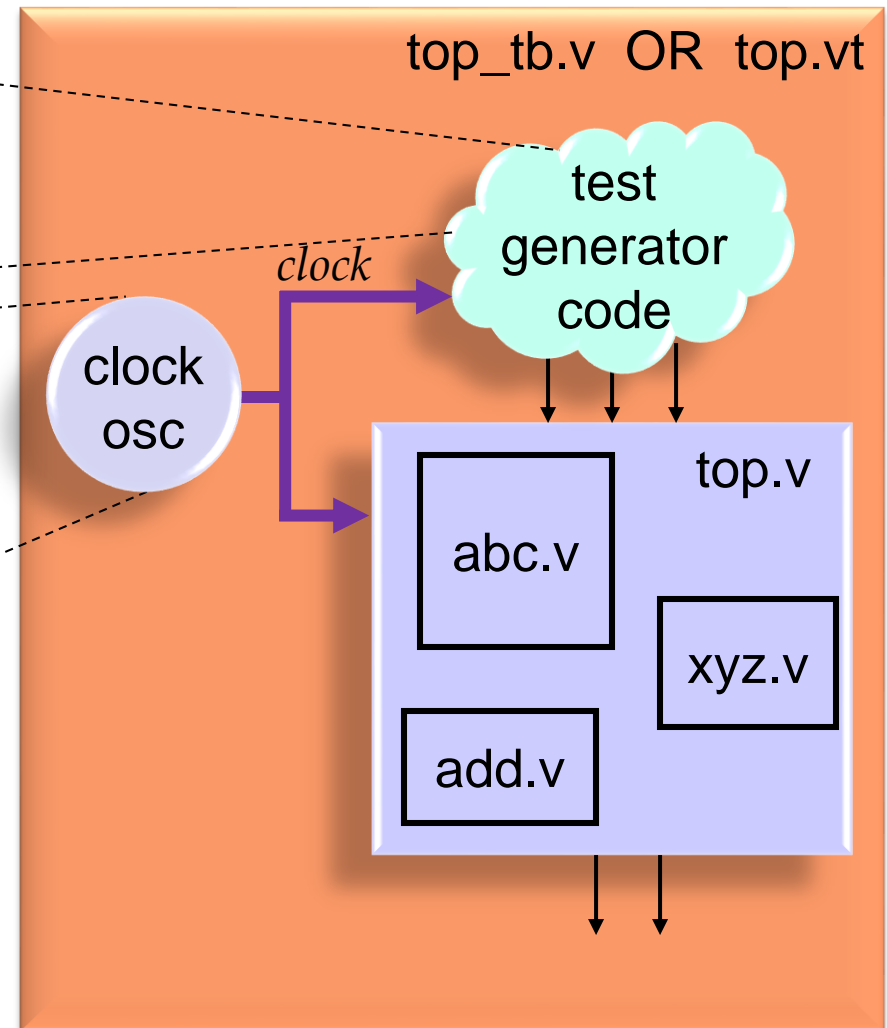
Example Clock Oscillator

```
reg clock;

initial begin
    reset = 1'b1;    // assert reset
    #1000;           // perhaps a few cycles
    reset = 1'b0;    // de-assert reset
    ...
    #10000;          // main simulation
    ...
    $stop;           // stop simulation
end

// osc inverts clock value every #100
always begin
    if (reset == 1'b1) begin
        clock = 1'b0;
        #1;          // let time advance when reset == 1'b1
    end
    else begin
        #100;        // cycle time = #200
        clock = ~clock; // invert clock
    end
end
```

- This code more cleanly sets *clock* in only one always block
- In this implementation, *reset* takes effect only at the end of a clock phase, not in the middle of one (synchronous reset of the clock)



Testbench Design: Approach 2

Example Test Generator

- Here is an example of how code in the test generator could look:

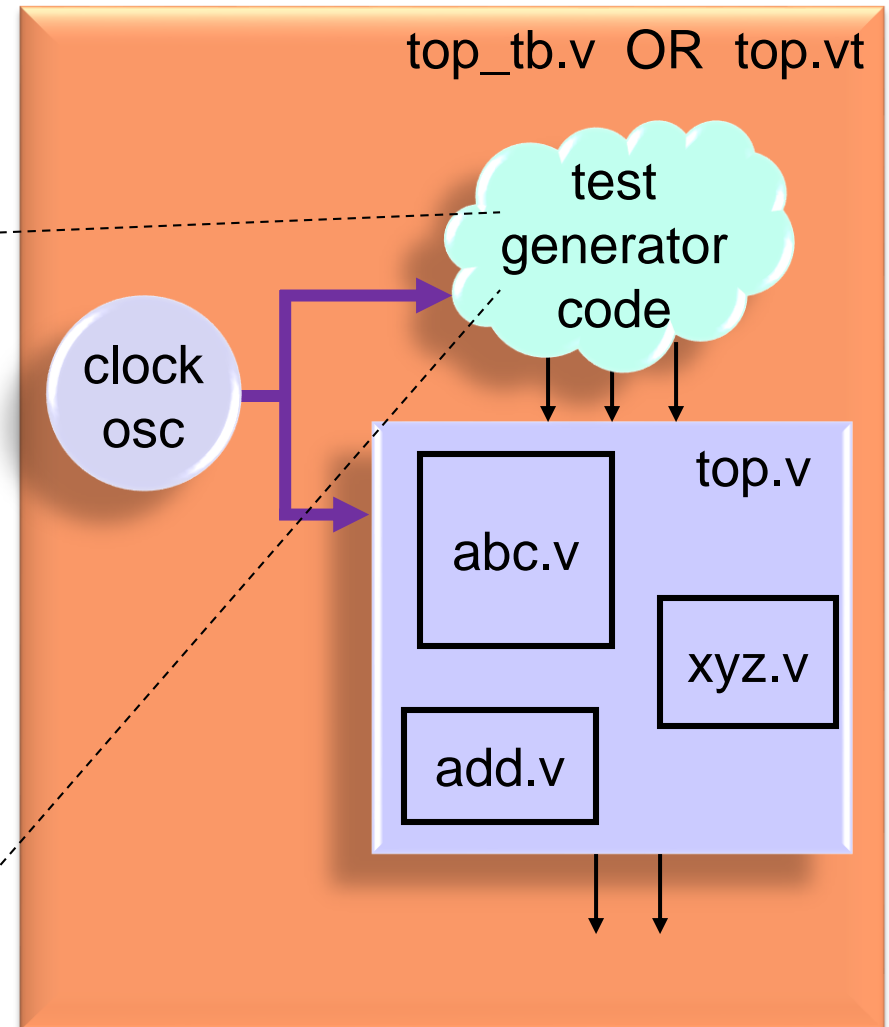
```
initial begin
  reset = 1'b1;           // set reset
  data  = 8'h00;
  #500;                   // wait > 1 clk period
  reset = 1'b0;           // clear reset

  // #10 after clk edge is clk-to-Q delay
  @(posedge clock); #10; // next clock edge
  data = 8'hB3;

  @(posedge clock); #10; // next clock edge
  data = 8'h0F;

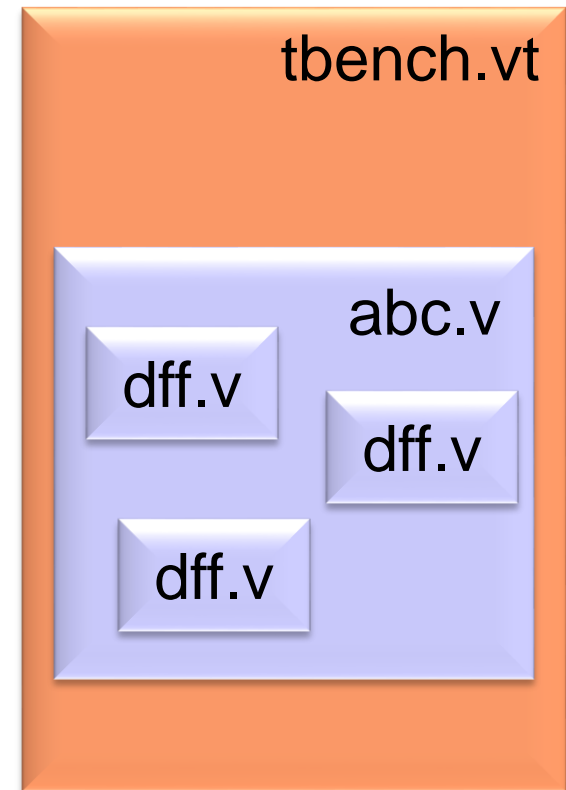
  @(posedge clock); #10; // next clock edge
  data = 8'hB7;

  @(posedge clk); #10
  @(posedge clk); #10
  repeat (50) @(posedge clk); // wait 50 clks
  $stop;                   // stop simulation
end
```



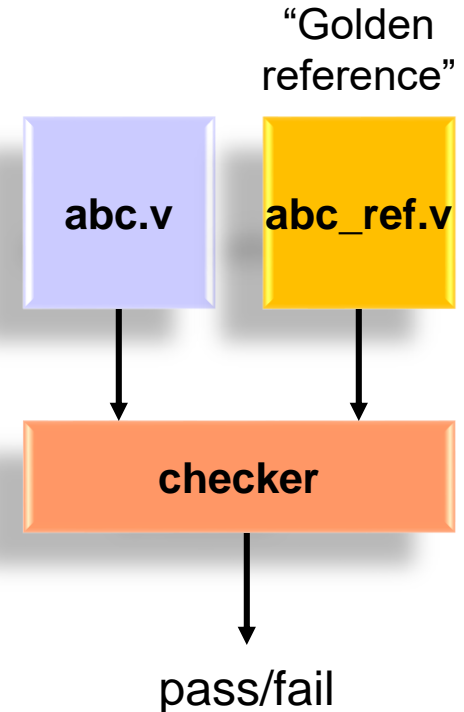
Example Given To You (EEC 281)

- Example test bench and modules on the course web page found with the link:
[Notes on running verilog](#)
- The example is not the most realistic partitioning (e.g., we would normally never put D FFs in their own modules), but it is a good working example to get you started
- Other files
 - **tbench.vf**
Contains files needed for simulator
 - **Makefile**
Very handy method for typing commands



Verifying Hardware Correctness

- A number of ways to verify designs:
 - 1) Eyeball text printouts
 - Quickest and easiest
 - 2) Eyeball waveforms
 - Quick and easy for some simple designs
 - 3) “Golden Reference” approach. Write target reference code and verify it matches your hardware design.
 - This is the most robust and is required for non-trivial designs
- As designs become more complex, verifying their correctness becomes more difficult



Verifying Hardware Correctness

3) The Golden Reference

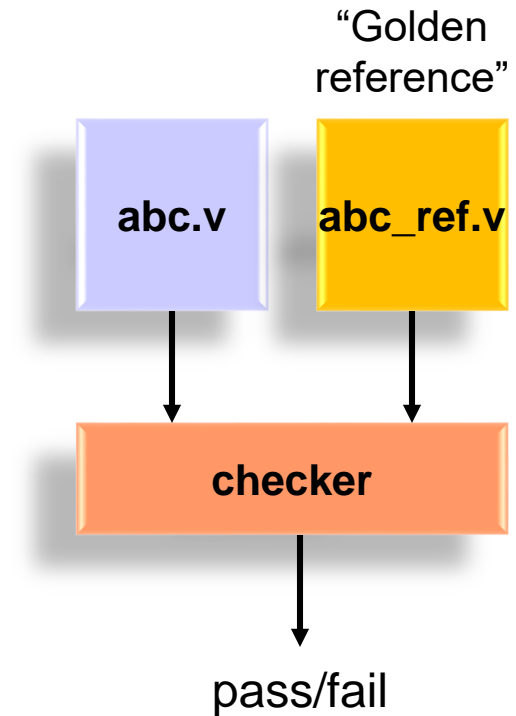
- Write an easy to understand simple model in a higher-level language
 - C or matlab commonly
 - Matlab is a natural choice for DSP applications
 - Must be written in *a different way* from the verilog implementation to avoid repeating the same bugs
- Designers agree the golden reference is the correct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

Comparing with the Golden Reference

- There are two major approaches to *comparing* with the Golden Reference:
 - A. 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
 - + Very easy to automate the comparison
 - + Likely less testing will be needed than approach (B)
 - The Golden Reference must now do awkward operations such as rounding and saturation that exactly match the hardware
 - For example, `floor(in + 0.5)` for rounding of 2’s complement numbers
 - B. Hardware and Reference must be “close”
 - + Golden Reference is simpler to write and has higher confidence
 - Inadequate for control hardware which must be a perfect match
 - Likely more testing will be needed than approach (A)
 - Example: calculate and compare the Signal-to-Noise Ratio (SNR) of the hardware vs. reference comparison
 - Comparisons could be complex and imperfect. For example, imagine how many ways two 5-minute audio signals could vary by 0.1% from each other. An imperceptible amplitude difference or a 3-second crash.

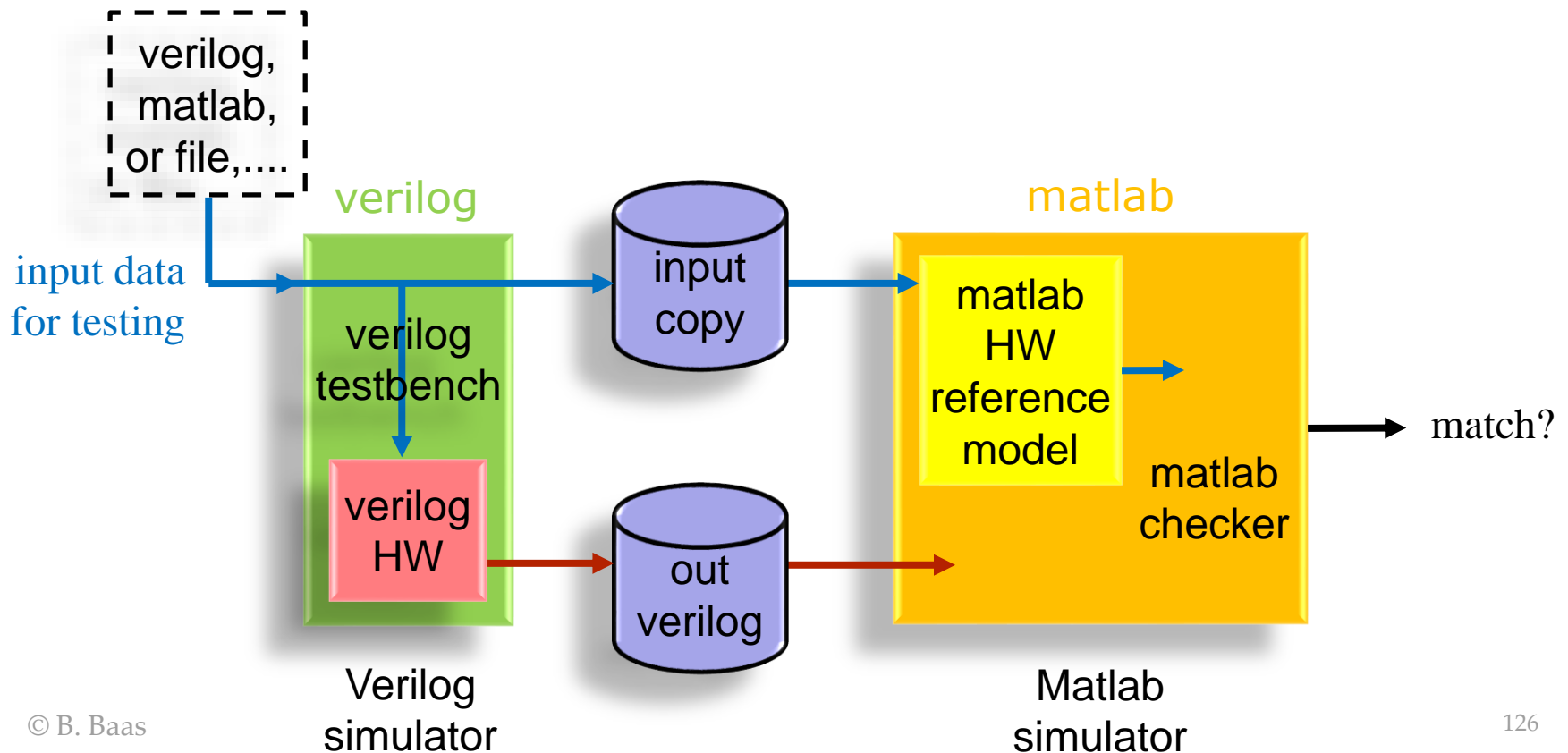
Golden Reference Approach: The Checker

- Implementing the “checker” comparison tool
 - A. Bit-accurate: comparisons could be done using:
 - the `verilog testbench` itself
 - `matlab`
 - the built-in “diff” command in `linux`
 - many other options
 - B. “Close enough” comparisons
 - `matlab` is a good place to look first
- To check the checker, at some point an error should be introduced in either the hardware design or the golden reference to verify the checker catches it



Golden Reference Approach: Example with Matlab Ref. and Checker

- Matlab is a fine choice for implementing the reference model and the checker



Golden References

With Different Levels of Detail

- A golden reference that uses a high-level function not written by you (e.g., `face_recognize.m`) will likely require a second version to be written which uses calculations you fully understand such as add, subtract, multiply, etc.
 - A High-level golden reference can be useful for early design exploration
 - A Low-level golden reference will be necessary to aid in hardware design

Golden Reference-**High** level

```
out = face_recognize(input);
```

Golden Reference-**Low** level

```
a = input + bias;  
b = a × scaling;  
c = convolve(b, coeffs1);  
...  
...  
out = round(m + k + 0.5);
```

Sources of Input Test Data

- 1) From a data file on disk
 - For example, from a video sequence
- 2) 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

```
% data.m
a(1) = 23;
a(2) = 456;
a(3) = 92;
a(4) = 4738;
...
```

Sources of Input Test Data

- Example data generated by verilog, then imported into matlab

```
% verilog_data.m
%
% Data file printed by a verilog simulation test bench.
%
% The data is printed in such a manner that the data
% may be loaded into matlab by simply typing in matlab:
%
% >> verilog_data

input(1) = -23;
input(2) = 2;
input(3) = -9;
input(4) = 93;
input(5) = 1;

a(1,1) = 5;
a(1,2) = 9;
a(1,3) = -1;
a(2,1) = 0;
a(2,2) = 5;
a(2,3) = -8;
a(3,1) = 4;
a(3,2) = -2;
a(3,3) = 5;
a(4,1) = 0;
a(4,2) = -1;
a(4,3) = -5;
```

```
>> verilog_data
>> whos
      Name      Size      Bytes  Class  Attributes

      a         4x3         96  double
      input     1x5         40  double

>> input

input =

    -23     2    -9    93     1

>> a

a =

     5     9    -1
     0     5    -8
     4    -2     5
     0    -1    -5

>>
```

Generating Test Cases

1) Exhaustive

- Example: a 16-bit adder has 32 inputs, so 2^{32} (4.3 billion) possible inputs. 71 minutes @ 1 million tests/sec
- Example: a 32-bit adder would require 584,942 years @ 1 million tests/sec!
- On the positive side, when you are done, you know your circuit is 100.000% correct

2) Directed—choose corner or edge cases by hand

- Example: 8-bit + 8-bit signed 2's complement adder
 - $0+0, 0+1, 1+0, 0+(-1), (-1)+0$
 - $(-1)+(-1) = 11111111 + 11111111$
 - $127+127 = 01111111 + 01111111$
 - $(-128)+(-128) = 10000000 + 10000000$

Generating Test Cases

3) Random

- The test environment automatically generates random input test cases, possibly with some direction
- It is almost always a good idea to make results repeatable to permit debugging of errors
 - Avoid: “it failed after a week of testing but now it works fine and I can not find the case that failed!”
 - Run short batches of tests
 - The random input data of each batch is determined by a random seed
- Run random tests on as much hardware as you can afford
- Run random tests as long as the schedule permits

Recommended Directory and File Layout (EEC 180)

