Design of Digital Circuits and Systems

Testing: Assertions, OOP

Instructor: Vikram Iyer

Teaching Assistants:

Ariel Kao
Selim Saridede
Derek Thorp
Josh Wentzien
Jared Yoder

Adapted from material by Justin Hisa

Relevant Course Information

- Quiz 4 this Thursday @ 11:**4**0 am
  - Algorithms to Hardware
- Lab 5 report due Friday (5/23)
- Lab 6 proposal due next week (5/28)
  - (1) Describe your major project behavior, features, components/modules, and user interaction in a few paragraphs
  - (2) Include at least a top-level block diagram (preferably with signals labeled on it; other diagrams welcome)
  - (3) Include images/sketches of VGA output
  - "Proposal Workshop" in lecture on 5/27

Testbenches

- HDL module that tests another module
  - Typically called the *device under test* (`dut`) or *unit under test* (`uut`)
  - No ports (*i.e.*, inputs or outputs)
  - Not synthesizable
  - Note: even if written in the same HDL, testbenches may give different simulation results on different simulators

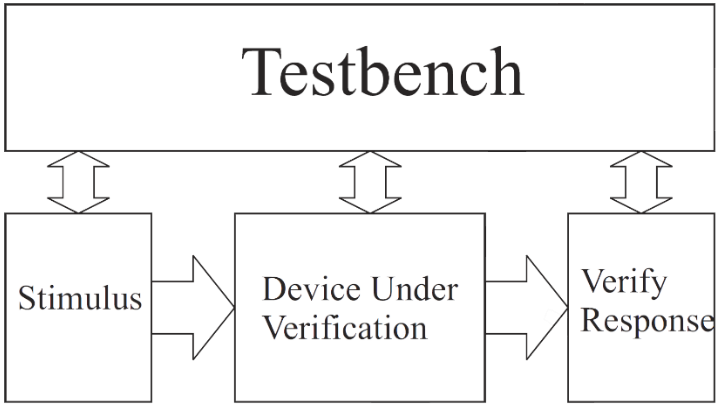


Figure 8.1: Modular testbench structure.

Test Vectors from a File

- Can be convenient to load test vectors from a file
  - Use `$readmemb` and `$readmemh`
  - Can also save you recompiling time! → can change values in file without recompiling

```
logic [W-1:0] test_vectors[0:15];   //array to hold test vectors

// define test inputs
integer i;
initial begin

    $readmemh("tests.txt", test_vectors);   could combine signals into single vector!
                                            {A,B,C} = test_vectors[i];
    Reset = 1; Start = 0;  @(posedge clk);
    Reset = 0;             @(posedge clk);
    for (i = 0; i < 2**4; i++) begin

        Start = 1; Num = test_vectors[i]; @(posedge clk);

        Start = 0;                        @(posedge Ready);
    end
    @(posedge clk);  // extra cycle of output
    $stop();
```

Dumping Responses

- The results of a simulation can be "dumped" to a file for later viewing in a waveform viewer or analysis
  - `$dumpfile` specifies the name of the file
    - `"dump.vcd"` by default (**V**alue **C**hange **D**ump)
    - Found in *`<Project>\simulation\modelsim`*
  - `$dumpvars` saves all of the variables from that point onward to that file
    - You can use arguments to specify which variables you want

```
// define test inputs
integer i;
initial begin

   $dumpfile("values.vcd");
   $dumpvars;

   Reset = 1; Start = 0;  @(posedge clk);
   Reset = 0;             @(posedge clk);
```

EDA Playground

- The advanced verification features we will discuss cannot be run in ModelSim so we will use EDA Playground instead
  - A web application that will let you use more powerful commercial simulators
  - Homework 6 will walk you through the registration process and a short tutorial
  - To use the waveform viewer in EDA playground, you *must* generate a `.vcd` file during your simulation! $dumpvars;

Checking Responses (Review)

- Visually checking simulated waveforms quickly becomes impractical for large designs simulated over thousands of clock cycles
  - Even for `isPrime`, we are constantly scanning right for `Done`, then scanning up and down for `P`.
  - Displaying and explaining your waveforms for labs has been tedious for a while now

- There are simulator-independent system tasks to write messages to the user/tester!
  - Look similar to `printf()` in C
    - `$<system_task>(<format_string>, <sig_1>, <sig_2>, …)`

Format Specifiers (Review)

Table 5.7: Format Specifiers.

Specifier	Meaning
%h	Hexadecimal format
%d	Decimal format (signed)
%o	Octal format
%b	Binary format
%c	ASCII character format
%v	Net signalstrength
%m	Hierarchical name of current scope
%s	String
%t	Time
%e	Real in exponential format
%f	Real in decimal format
%g	Real in exponential or decimal format

\ escape character

Table 5.8: Special characters.

Symbol	Meaning
\n	New line
\t	Tab
\\	\ character
\"	" character
\xyz	Where xyz is are octal digits - the character given by that octal code
%%	% character

- Warning: these differ from the specifiers for `printf`
- The *minimum* field width is specified by numbers between the '%' and specifier letter
  - *e.g.*, `%3d` will pad out to 3 digits if necessary, `%0d` will show just the minimum number of digits needed

Checking Responses: `$display` (Review)

- Triggers once when encountered, prints the given format string and adds a new line:

```
// define test inputs
integer i;
initial begin
   Reset = 1; Start = 0;  @(posedge clk);
   Reset = 0;             @(posedge clk);
   for (i = 0; i < 2**W; i++) begin
      Start = 1; Num = i; @(posedge clk);
      Start = 0;          @(posedge Ready);

      $display("T = %4t, isPrime(%2d) = %s",
               $time, Num, P ? "Yes" : "No
");

   end
   @(posedge clk);  // extra cycle of output
   $stop();
end
```

current simulation time

```
Transcript
VSIM 4> run -all
# T =   90, isPrime( 0) = No
# T =  150, isPrime( 1) = No
# T =  210, isPrime( 2) = Yes
# T =  270, isPrime( 3) = Yes
# T =  330, isPrime( 4) = No
# T =  410, isPrime( 5) = Yes
# T =  470, isPrime( 6) = No
# T =  570, isPrime( 7) = Yes
# T =  630, isPrime( 8) = No
# T =  710, isPrime( 9) = No
# T =  770, isPrime(10) = No
# T =  910, isPrime(11) = Yes
# T =  970, isPrime(12) = No
# T = 1130, isPrime(13) = Yes
# T = 1190, isPrime(14) = No
# T = 1270, isPrime(15) = No
```

Checking Responses: `$write`

- Triggers once when encountered, prints the given format string *without* a new line:

```
// define test inputs
integer i;
initial begin
   Reset = 1; Start = 0;  @(posedge clk);
   Reset = 0;             @(posedge clk);
   for (i = 0; i < 2**W; i++) begin
      Start = 1; Num = i; @(posedge clk);
      Start = 0;          @(posedge Ready);

      $write("T = %4t, isPrime(%2d) = %s\n",
             $time, Num, P ? "Yes" : "No ");

   end
   @(posedge clk);  // extra cycle of output
   $stop();
end
```

Same messages? Yes

Transcript

```
VSIM 3> run -all
# T =   90, isPrime( 0) = No
# T =  150, isPrime( 1) = No
# T =  210, isPrime( 2) = Yes
# T =  270, isPrime( 3) = Yes
# T =  330, isPrime( 4) = No
# T =  410, isPrime( 5) = Yes
# T =  470, isPrime( 6) = No
# T =  570, isPrime( 7) = Yes
# T =  630, isPrime( 8) = No
# T =  710, isPrime( 9) = No
# T =  770, isPrime(10) = No
# T =  910, isPrime(11) = Yes
# T =  970, isPrime(12) = No
# T = 1130, isPrime(13) = Yes
# T = 1190, isPrime(14) = No
# T = 1270, isPrime(15) = No
```

Checking Responses: `$monitor`

- Triggers when encountered, then triggers anytime one of its signal changes (adds a new line):

```
// define test inputs
integer i;
initial begin
   $monitor("T = %4t, isPrime(%2d) = %s\n",
            $time, Num, P ? "Yes" : "No ");

   Reset = 1; Start = 0;  @(posedge clk);
   Reset = 0;             @(posedge clk);
   for (i = 0; i < 2**W; i++) begin
      Start = 1; Num = i; @(posedge clk);
      Start = 0;          @(posedge Ready);
   end
   @(posedge clk);  // extra cycle of output
   $stop();
end
```

triggers when Num or P changes

Same messages? No

Transcript

```
VSIM 6> run -all
# T =    0, isPrime( x) = He
# T =   30, isPrime( 0) = He
# T =   70, isPrime( 0) = No
# T =   90, isPrime( 1) = No
# T =  150, isPrime( 2) = No
# T =  190, isPrime( 2) = Yes
# T =  210, isPrime( 3) = Yes
# T =  270, isPrime( 4) = Yes
# T =  310, isPrime( 4) = No
# T =  330, isPrime( 5) = No
# T =  390, isPrime( 5) = Yes
# T =  410, isPrime( 6) = Yes
# T =  450, isPrime( 6) = No
# T =  470, isPrime( 7) = No
# T =  550, isPrime( 7) = Yes
# T =  570, isPrime( 8) = Yes
# T =  610, isPrime( 8) = No
# T =  630, isPrime( 9) = No
# T =  710, isPrime(10) = No
# T =  770, isPrime(11) = No
# T =  890, isPrime(11) = Yes
# T =  910, isPrime(12) = Yes
# T =  950, isPrime(12) = No
# T =  970, isPrime(13) = No
```

garbage

Lecture Outline

- Testbenches (yet again)
- **Assertions**
- Object-Oriented Programming

Assertion-Based Verification

- `$display`, `$write`, `$monitor`
  - Can indicate the response of the circuit in textual form
  - Still must be *verified manually/visually*, even if you also print the expected response alongside it

- **Assertions** are SystemVerilog features that can print messages when an expected condition fails
  - `assert` – *immediate* assertion that follows simulation event semantics
  - `assert property` – *concurrent* assertion based on clock semantics

Immediate Assertions

- An **immediate assertion** is an if-else statement with a default-generated else:

```
assert (P == 1);
```

↔

```
if (P == 1);   // nothing if true
else $error("Assertion error.");
```

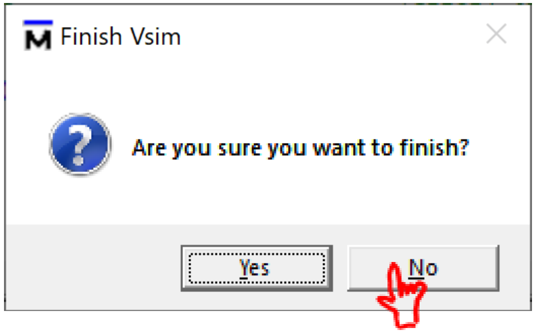
  - Must be contained inside of a procedural block

- Can also explicitly define *pass* and *fail* statements:

```
// defined pass, default fail
assert (P == 1) $display("%2d is prime", Num);

// default pass (nothing), defined fail
assert (P == 1) else $error("%2d is not prime", Num);

// defined pass, defined fail
assert (P == 1) $display("%2d is prime", Num);
else $error("%2d is not prime", Num);
```

Failure Messages

- Messaging: `$info`, `$warning`, `$error`
  - Ordered in increasing severity (less severe are suppressible)
  - Same argument format as `$display`, `$monitor`
  - All print additional debugging line (time, scope, file, line), but simulation continues
- Break: `$fatal`
  - Takes an `error_code` as extra (1st) argument that is passed to `$finish`, which terminates the simulation
  - ModelSim produces this pop-up box:
    - Click "No", otherwise ModelSim will exit



Short Tech Break

Concurrent Assertions

- **Concurrent assertions** run continuously throughout simulation based on a sampling clock and can test for much more complex behaviors
  - Do not need to be placed inside another procedural block
  - Assert that a specified *property* is true
  - Like immediate assertions, can specify pass/fail code
  - *Unfortunately, these do not work in ModelSim* (need a more fully-featured simulator)
- Example: assert that Ready and Done are never true at the same time

```
property ready_nand_done;
   @(posedge clk) ~(Ready & Done);
endproperty
assert property (ready_nand_done);
```

property definition (lines 1–3); assertion (line 4); pass code; else // fail code

Properties

- Defined between property and endproperty
  - Includes the ability to define an argument list!
    - *e.g.*,

```
property Nand(logic A, logic B);
   @(posedge clk) ~(A & B);
endproperty
assert property (Nand(Ready, Done));
```

  - Can be defined in-line, but this is stylistically discouraged

- Complex properties are typically active over (*i.e.*, they span) a period of time
  - Specified using a combination of *implications* and *sequences*
    - *e.g.*,

```
property handshake;
   @(posedge clk) Req |-> ##[1:2] Ack;
endproperty
```

Ack should be asserted 1-2 clock cycles after Req

Implications (Mathematics)

- $p \Rightarrow q$ is read as "p implies q"
  - A statement meaning: if p is true, then q must also be true
  - The statement evaluates to true or false based on whether the actual values of p and q *support* the implication:

p	q	$p \Rightarrow q$
false	false	true
false	true	true
true	false	false
true	true	true

no way to contradict the implication (first two rows)

← contradicts the implication (third row)

- Logically equivalent to `!`p `||` q or p `?` q `:` `1`

Implications (SystemVerilog)

- **Implications** are notated by A |-> C and A |=> C
  - A is the *antecedent* (LHS), C is the *consequent* (RHS)
  - The consequent is only evaluated if the antecedent is true
  - In the context of assertions and properties, evaluating to true is a pass and false is a fail
- Implication timing:
  - An overlapped implication (|->) evaluates C in the *same* clock cycle that A was true
  - A non-overlapped implication (|=>) evaluates C on the *next* clock cycle after A was true
- Practice: write an equivalent implication to ~(A&B)

A	B	NAND
0	0	1
0	1	1
1	0	1
1	1	0

p, ~q → (row 1 1 0)

evaluated simultaneously, so overlapped: A |-> ~B

Sequences

- A **sequence** is a series of Boolean expressions with defined relationships *in time*
  - Any Boolean expression is, by itself, an implicit sequence
  - Sequences can be constructed from other sequences and *sequence operators*
  - You can name a sequence and give it arguments using `sequence` and `endsequence`
- Common sequence operators:
  - $\#\#N$ – delays next sequence by N cycles
  - $[\star N]$ – N consecutive repetitions of the LHS
  - $[=N]$ – N non-consecutive repetitions of the LHS
  - Any N can be replaced by the inclusive range $A:B$

Req |-> ##[1:2] Ack
A |-> ##3 B
A |=> B [*3]
A |-> B [=3]

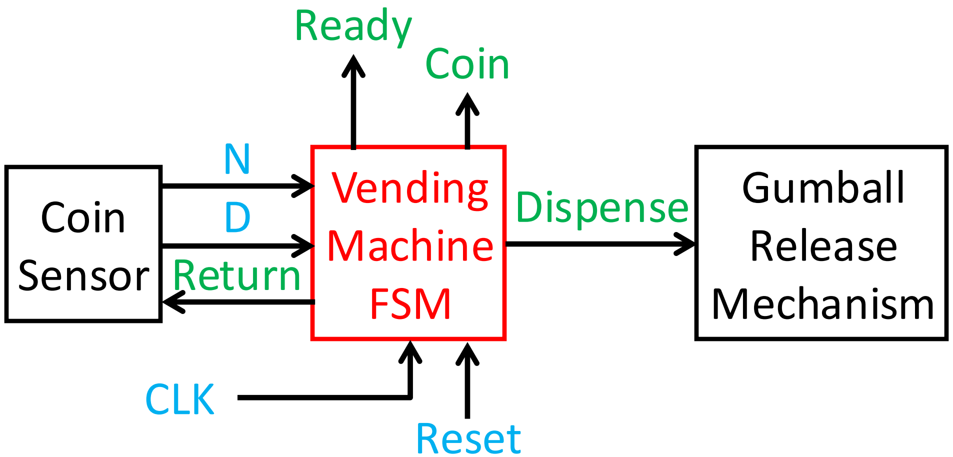
Sequences

- Example: rewritten handshake property

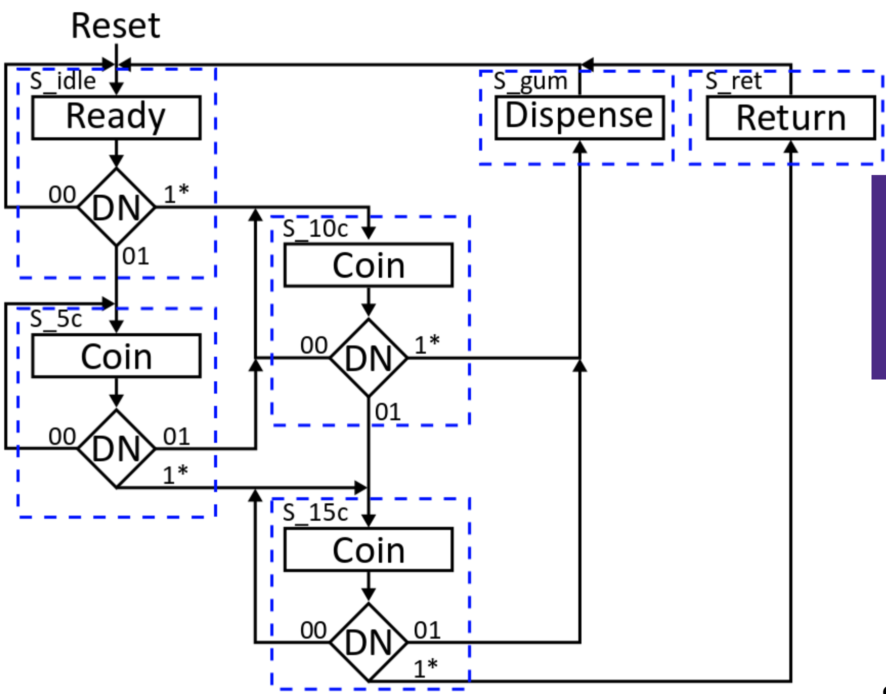
```
sequence request;
   Req;
endsequence

sequence acknowledge;
   ##[1:2] Ack;
endsequence

property handshake;
   @(posedge clk) request |-> acknowledge;
endproperty
```

Assertion Example

- Modified vending machine specs:
  - The machine only accepts dimes (D, 10¢) and nickels (N, 5¢)
  - Once 20¢ has been inserted, a gumball is dispensed; if more than 20¢ is inserted, all coins are returned
  - The machine has two lights
    - One to show that it is ready for the next transaction (`Ready`)
    - One to show that further coins need to be inserted (`Coin`)



Vending Machine ASM Chart & State Table



State	Next State D	Next State $\overline{D}N$	Next State $\overline{D}\,\overline{N}$	Ready	Dispense	Return	Coin
S_idle	S_10c	S_5c	S_idle	1	0	0	0
S_5c	S_15c	S_10c	S_5c	0	0	0	1
S_10c	S_gum	S_15c	S_10c	0	0	0	1
S_15c	S_ret	S_gum	S_15c	0	0	0	1
S_gum	S_idle	S_idle	S_idle	0	1	0	0
S_ret	S_idle	S_idle	S_idle	0	0	1	0

Testing the Vending Machine

- `Dispense` and `Ready` should never be asserted at the same time
  - Write an *immediate* assertion to double-check this fact in an `always` block:

```
always @(*)
    assert (~(Dispense & Ready))
    else  $error("Dispense and Ready both asserted");
```

  - Now write a *concurrent* assertion to double-check this fact on each clock edge:

```
property DispenseNandReady;
    @(posedge clk) ~(Dispense & Ready);
end property

assert property (DispenseNandReady)
else  $error("Dispense and Ready both asserted");
```

Testing the Vending Machine

- Write properties to double-check the following expected behaviors:
  - From the idle state, inserting a coin (D, N) should cause the `Coin` output to be asserted:

```
property idle_to_coin;
  @(posedge clk) (ps == S_idle) & (D | N) |=> Coin;
endproperty
```

(ps == S_idle) can replace with: Ready

- Scope reminder:
  - You may want to express an immediate assertion or property using states (`parameter`, `enum`)
  - Make sure that the assertion or property is inside the appropriate module then (not the test bench)

Testing the Vending Machine

- Write properties to double-check the following expected behaviors:
  - In every clock cycle, exactly 1 of `Ready`, `Coin`, `Dispense`, and `Return` should be asserted:

messy using regular logic: any of 0001, 0010, 0100, 1000

many convenient system tasks exist to help with writing properties:

```
property outputs;
    @(posedge clk) $onehot({Ready, Coin, Dispense, Return});
endproperty
```

Aside: Default Clocking

- Instead of putting the clock edge in every property, it is possible to define a default clocking block:

```
default clocking clock_block;
   @(posedge clk)
endclocking
```

  - Then you can omit the `@(posedge clk)` clause in properties and assertions!

Short Tech Break

Lecture Outline

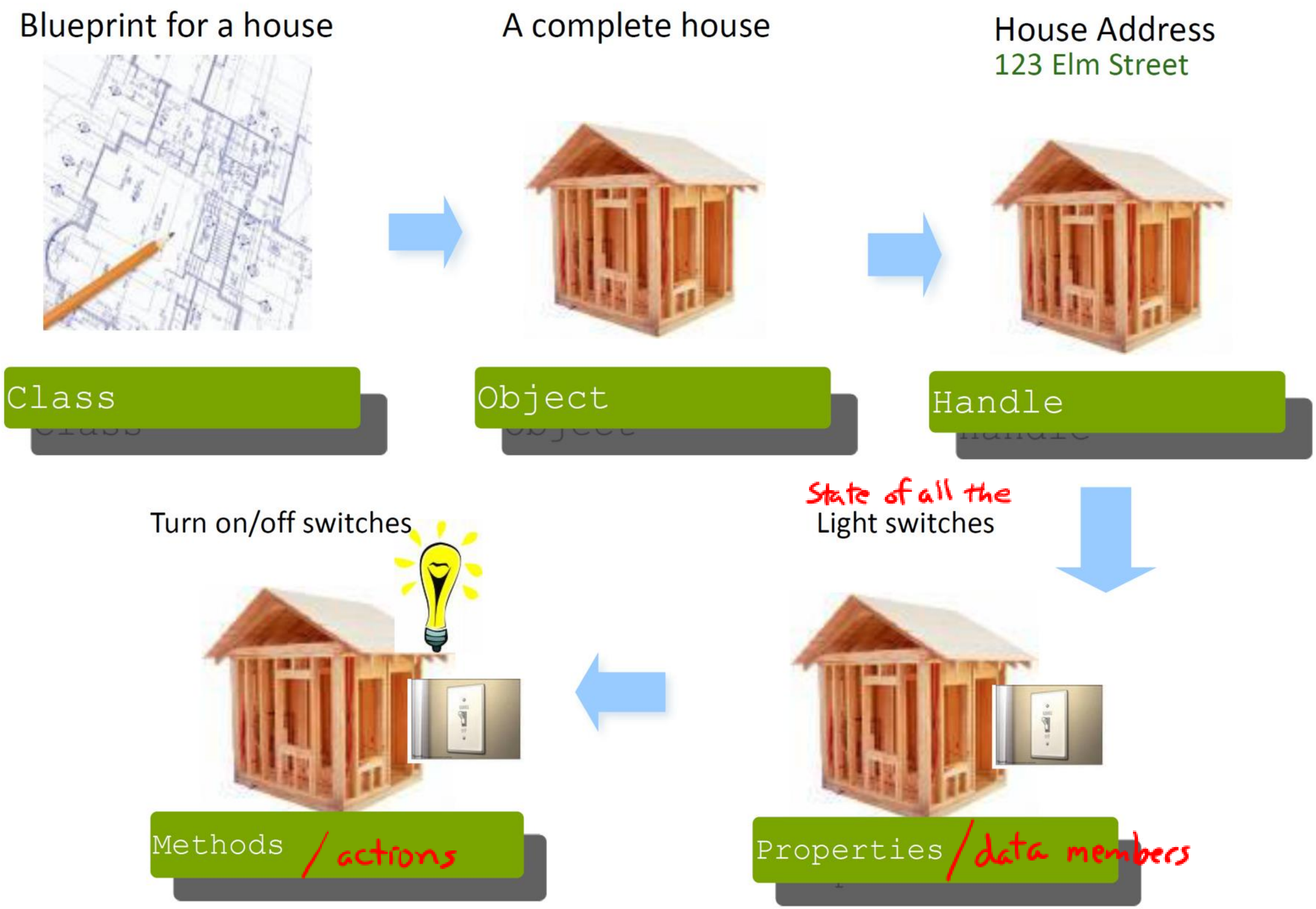
- Testbenches (yet again)
- Assertions
- **Object-Oriented Programming**

Object-Oriented Programming

- SystemVerilog allows for OOP
  - Including inheritance and polymorphism
  - *For verification – not synthesizable (no good in ModelSim)*
- Encapsulates the data together with the code/routines that manipulates them
  - Proper usage can yield gains in productivity, maintainability, and thoroughness
- Facilitates testing – testbench's goal is to apply stimuli and then check to see if the result is correct
  - We can model our testbenches as objects that perform a sequence of actions: create a **transaction**, transmit it, receive the result, check the result, report any issues

(need a more fully-featured simulator)

OOP Terminology



Defining a Class

- A class is defined between `class` and `endclass`

```
class Transaction;

   bit [31:0] addr;

   function void display();
      $display("Transaction: %h", addr);
   endfunction

endclass
```

- Can be defined at the top-level or within a `module` or `package`
  - Typically define each class in a separate file, or can group related classes in packages

Aside: Packages

- A package creates an explicitly named scope that contains declarations intended to be shared
  - Can contain types, variables, tasks, functions, sequences, properties, classes, etc.
  - Must be a top-level block

```
package pack;
   class Trans;
      // class body
   endclass
endpackage
```

- Package components can be accessed directly via the scope resolution operator (::) or imported

```
module use_trans();
   initial begin
      pack::Trans tr;
      // test code
   end
endmodule
```

could also just import pack::Trans

```
module use_trans();
   import pack::*;
   initial begin
      Trans tr;
      // test code
   end
endmodule
```

everything

Constructing and Using Objects

- ① Create class handle, ② instantiate an object instance, ③ use dot notation to access properties and methods:

```
module use_trans();
   initial begin
      // separate
   ①  pack::Trans tr;
   ②  tr = new();
   end
endmodule
```

```
module use_trans();
   initial begin
      // combined
   ①②  pack::Trans tr = new();
   ③   tr.display();      argument list
   ③   $write("%0d", tr.addr);
   end          method name    property name
endmodule
```

- Can define/override the class constructor:

```
class Transaction;
   bit [31:0] addr;

   function new();      special method
      addr = 371;
   endfunction

   // rest of class definition...
```

Classes Exercise

- A `MemTrans` class to generate transactions for memory modules
- Create the class with the following:
  - `data_in` property of logic type (8 bits)
  - `addr` property of logic type (4 bits)
  - `write` property of logic type (1 bit)
  - `void` function that prints out the values of `data_in` and `addr` in hex and `write` in binary → $display
  - A reasonable constructor → set to all zeros?
- Create a `mem_test` module that instantiates a `MemTrans` object and invokes its function } put in an initial block

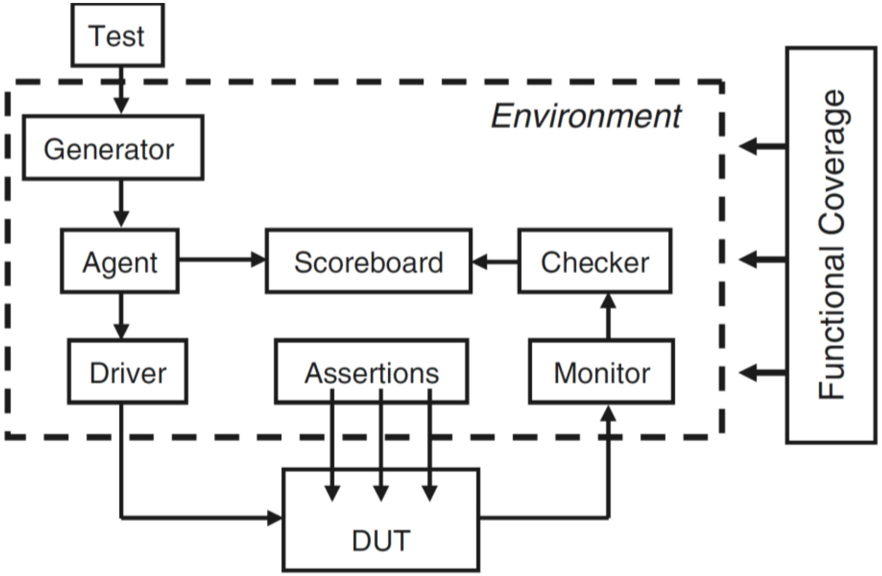
Classes Exercise Sample Solution

```
class MemTrans;
  logic [7:0] data_in;
  logic [3:0] addr;
  logic write;

  function void print();
    $display("data_in = 0x%2h", data_in);
    $display("addr = 0x%h", addr);
    $display("write = %b", write);
  endfunction

  function new();
    {data_in, addr, write} = 13'd0;
  endfunction
endclass
```

```
module mem_test ();
  MemTrans tr;
  initial begin
    tr = new();
    tr.print();
  end
endmodule
```

Layered Testbenches

- Each block is an object and passes transaction objects
  - *Generator* creates transactions
  - *Driver* talks to design
  - *Monitor* receives response
  - *Scoreboard* compares response to expectations



- Transactions can be transferred and held in FIFO buffers for queuing

Looking Ahead

- Classes are required for SystemVerilog's constrained **randomization** features

- Randomized testing
  - Difficult to completely test large designs
  - Can be hard to anticipate all edge cases
  - Want to find unexpected errors
    - Designed tests only cover what you are anticipating