Documentation Guidelines

Laboratory Reports

All laboratory reports will have a cover page with the students' names, the class, the problem number and the problem title prominently displayed.

All lab reports must include printouts of graphical waveforms showing simulation results and log files of the simulations done. Resolve any errors or warnings before submitting a lab report. **Submitted log files may not contain warning or error messages.** All text, code and waveforms are to be computer printed without edits. You may copy the file into a word processor adding page breaks, titles and page numbers but nothing more.

While these computer-generated outputs and all source code must be included in a lab report, they are supporting documentation, not the body of the lab report. The essential elements of a lab report are:

- A statement of what you are accomplishing, demonstrating, determining, etc.
- Methodology: statement of how you did it. Specifically include a test plan. Include flow charts and other materials as appropriate.
- Analysis of results: what did you demonstrate/prove.
- Conclusions.

Any lab report without the above elements will be incomplete and will be graded accordingly.

Academic Dishonesty

Submitting any report that is not entirely your own work is a form of academic dishonesty and will not be tolerated. Each and every lab report must include the following statement, signed and dated by the student. Lab reports without the statement will be summarily rejected.

I hereby attest that this lab report is entirely my own work. I have not copied either code or text from anyone, nor have I allowed or will I allow anyone to copy my work.

Name (printed) ______________________________

Name (signed) ______________________________ Date ________________

Documentation

This is a course about hardware development using a hardware description language. The application may be modeling hardware but the method is coding of language constructs. As such, **documentation of your circuit descriptions will be a key part of the grading**. All modules should be clearly written and well documented.

The purpose of documentation is to identify the code and to explain both the "what" and the "why" of the code. In school, documentation is also where you demonstrate your understanding of class material. Part of learning the skill of documentation is knowing what is important and what is trivial. Do not document the trivial. Over the course of the class, what is new and important in the early problems may become trivial later on.

Each module should have a header that identifies it and its author(s). An example is:

```
/**************************************************************************
 ***                                                                    ***
 *** EE 526 L Experiment #1                 Student_Name, Spring, 2003 ***
 ***                                                                    ***
 *** Experiment #1 title                                                ***
 ***                                                                    ***
 **************************************************************************
 ***  Filename: Filename.v              Created by Student_Name, Date  ***
 ***    --- revision history, if any, goes here ---                    ***
 **************************************************************************/
```

Module headers **must** give the filename, the author(s) and the experiment number.

You will write and debug modules without documenting each change. But once submitted, further changes should be noted. There will be such cases: a clock module, used with one problem, then modified and used with another problem. That second use may require several changes but only the overall change should be documented.

Normal practice, whether for this course or in professional engineering environments, is to put each module in a separate file, although it is perfectly legal to combine modules in one file. Whether separate or combined, each module must have a module header that describes it.

Module Purpose

After the header, document the purpose of the module if it's not obvious. A module that models hardware should describe what it models and anything special or important about how the modeling is done. A module that tests another module or modules (called a *testbench*) should document what it tests and what its test strategy is.

Test Plan

A test plan is a high-level description or "executive summary" of the tests performed by the module. It is not an exhaustive list of all the tests being performed. Every lab report must include a test plan documenting what tests are to be performed and explaining why they are adequate for the device under test. For most labs, an exhaustive test strategy is impractical, so some thought will have to be put into devising an adequate yet non-exhaustive test plan. It is extremely unlikely that a test plan that does not cause every circuit output to change states at least once will be satisfactory.

An example test strategy for testing an 8-input MUX might be:
"Test strategy: Test A-path (all ones, all zeros and alternate bits), B-path, undefined and hi-z select, example undefined and hi-z on A- and B-path inputs."

An example test strategy for testing an adder might be:
"Test strategy: Apply all 0 and 1 cases, with and without carry-in to test carry propagation (or no-carry propagation); apply alternate 1-0 and 0-1 cases (each cell adds without carry-out); apply alternate 1-0 and 0-1 cases (each cell adds with carry-out); add max positive and max negative with and without carry-in to test for arithmetic overflow."

An alternate test strategy for the adder might be:
"Test strategy: Generate exhaustive test of all combinations of X- and Y-inputs and carry-in."

When the purpose of a test is not obvious, add a parenthetical comment explaining what is being tested (*e.g.* carry propagation).

Model Documentation

Where different modeling approaches are possible, explain your choice. For example, there are different ways of assigning values to variables and they have subtle differences in their operation. Your documentation should indicate your understanding of the differences and why the chosen construct is the correct choice for this particular model. Any of the following cases require explanation (and this is not an exhaustive list!):

1. Use of an intra-assignment delay.
2. Use of a non-blocking assignment.
3. Use of a clocking construct (**posedge** or **negedge**) with a signal other than a clock or reset.
4. Use of a **wait** delay.
5. Use of a procedural continuous assignment with a signal other than a reset or preset.
6. Use of multiple procedural continuous assignment statements with a single signal.
7. Handling of exception conditions such as simultaneously active read and write strobes or arithmetic overflow.

In industry, there is a strong bias for reusing modules. This requires that each module document any limitations or unusual behavior. This applies in the lab, too. If your model is incomplete in some fashion, whether by hardware limitation or abbreviated coding you must identify the limitations that flow from that incompleteness. For example, an adder module may have no signal to indicate arithmetic overflow. You must document this in your module.

Modeling Style

Modules have different sections and different functions. Especially for the first few problems, you should identify each section of each module: port or signal declarations, netlists, module instantiations, signal display, test stimuli, etc. In all cases, you should use a blank line or two to separate sections. When modeling sequential logic, it's usually a good idea to model synchronous and asynchronous functions separately.

```
// Describe the asynchronous behavior
        code            // reset is active low

// Describe the synchronous behavior
        code            // clock is positive edge triggering
```

Use indentation to indicate the level or importance of each line of code. Indentation improves the look of the code, organizes it and because delimiters should pair at the same level of indentation it aids in troubleshooting.

A TAB indents too far so use three or four spaces for each level of indentation. All code between the **module** and **endmodule** keywords is subordinate to the module and should be indented one level. Code outside the module (typically **`timescale** or **`define**) is at the same indentation level as the **module** keyword.

Some constructs have subordinate clauses or subprocesses. These should be indented below the superior process. For example:

```
initial
        initialization_action;          // indented one level below keyword

if (test_clause)
        action_if_test_is_true;         // indented one level below keyword
```

Other constructs serve to delimit sections of code. The most common is the **begin…end** keyword pair. Others include **fork… join** and **case… endcase**. All code within these delimiters should be indented one level below the delimiter. The delimiters themselves may also be indented as shown in the following examples:

```
case ({sel1, sel0})
    0: action_for_selection_zero
    1: action_for_selection_one
    2: action_for_selection_two
    3: action_for_selection_three
    default: action_for_undefined_selections
endcase


if (test_clause)
    begin
        true_action_one;
        true_action_two;
    end
else
    begin
        false_action_one;
        false_action_two;
    end
```

Lines that are too long to fit comfortably on a page, *i.e.* longer than about 75 or 80 characters, can be broken anywhere there is whitespace, typically right after a comma. The rightmost part of the line is indented at its proper level. The remaining part or parts of the line are indented one more level.

Strings are an exception to the "break the line at whitespace" rule. Strings cannot continue on a second line. They must be broken into two separate strings, each on its own line and (usually) separated by a comma. Strings can, however, be broken anywhere except between the backslash and following character of so-called escaped characters.

```
"A long string cannot be broken somewhere in the middle and continued
    on another line."

"But a long string can be broken into two strings, one on one line, a comma,",
    " and the other on the following line."
```

Very occasionally (in this class, at least), a statement is both long and deeply indented. Breaking such a line according to the usual procedure results in many short, choppy lines that are difficult to read. In such cases, *outdent* or move up (left) in indentation levels so that more of the statement can appear on each line. As a rule of thumb, outdent at least two levels. Of course the outdent is used only for the extremely long statements and should not be continued with later statements.

```
module
    o    o    o
```

```
                              begin
Short_statement_isn't_outdented
                                            Extremely_long_statement_deeply
                                                 indented_and_broken
                                                 according_to_usual
                                                 procedure_resulting_in
                                                 many_short_and_choppy
                                                 lines
              Extremely_long_statement_deeply_indented_then_outdented
                   So_more_of_the_statement_appears_on_each_line_thus
                   Making_the_statement_easier_to_read
              Another_extremely_long_statement_deeply_indented_then_outdented
                   so_more_of_the_statement_appears_on_each_line
Short_statement_isn't_outdented
                                            Short_statement_isn't_outdented
                                            Short_statement_isn't_outdented
                              end
              o      o      o
     endmodule
```

Commonly the **$monitor** and **$display** statements, while not deeply indented, will still extend far beyond the right margin of the page. These statements can be broken anywhere there is a comma (except within a string):

```
     $display ("----Time----  ----Address----  ----Data ----  Rd  Wr  ALE  M/IO",
          "  s0  s1  NMI  TxD  RxD");
     $monitor ($time, %hAdr  %hData   %b   %b   %b   %b  ",
     "%b  %b  %b  %b  %b", MemAddr, MemDat, Read_Not, Write_Not, ALE, Mem_IO, Status0, Status1, NMI,
     TxData, RxData);
```

The printout generated by a monitor or display statement may also be longer than a printable line. Such lines can be compacted or indented with the "\n" and "\t" constructs, signifying newline and tab, respectively.

For all submitted printouts, all lines must fit. <u>Any lines exceeding the printable width (and therefore not available for review) will be penalized</u>. It is the student's responsibility to ensure that the full text of any line is printed.

Verilog allows variable names to be essentially unlimited in length. It is therefore advisable to assign descriptive variable names wherever possible (clock, select, data_in, etc.). On the other hand, someone must type these long variable names and every additional character increases the chance of a typo, so some restraint is necessary. One exception is loop indices where single letters, typically *i*, *j*, *k*, etc. are common and well understood.

How to Lose Points

As the semester progresses, inefficient or unnecessary code will be penalized since it shows a lack of understanding of how Verilog works and since it violates the *sine qua non* of engineering: elegance of design.

Strive for an aesthetically pleasing appearance as well as for completeness and technical accuracy.

Example of a poorly written module:

```
module mux2(x,y,s,e,z);
output z;
input x, y, s, e;
always @ (x or y or s or e) if (e) z = 1'bz; else z = s?y:x;
endmodule
```

Example of a well written (and documented) module:

```
/***********************************************************************
 ***                                                               ***
 *** EE 526 L Experiment #1                  Student_Name, Spring, 2003 ***
 ***                                                               ***
 *** Experiment #1 title                           Group # group_number ***
 ***                                                               ***
 ***********************************************************************
 ***  Filename: Filename.v               Created by Student_Name, Date  ***
 ***    --- revision history, if any, goes here ---                   ***
 ***********************************************************************
 *** This module models a 2:1 mux with active low enable:             ***
 ***    Enable      Select       Mux_Out                              ***
 ***      1       don't care       Hi-Z                               ***
 ***      0           1          B_Input                              ***
 ***      0           0          A_Input                              ***
 ***      0       undefined      A_Input                              ***
 *** The undefined Enable case is not modeled.  Module treats undefined  ***
 *** Enable as if it were zero (enabled).                             ***
 ***********************************************************************/

      module mux2(A_Input,    // LS mux input
                  B_Input,    // MS mux input
                  Select,     // Select signal (see truth table above)
                  Enable,     // Mux enable, active low
                  Mux_Out);   // Mux output

      // Declare the inputs and outputs
            output Mux_Out;
            input A_Input, B_Input, Select, Enable;

      // Declare signal type
            wire Mux_Out, A_Input, B_Input, Select, Enable;

      // Model the mux
            always @ (A_Input or B_Input or Select or Enable)
                  if (Enable)
                        Mux_Out = 1'bz;    // disabled case
                  else
                        if (Select)
                              Mux_Out = B_Input;
                        else
                              Mux_Out = A_Input;

      endmodule
```

Testing and Verification of Digital Logic

The purpose of testing is to develop confidence that a design or its specific implementation is functioning correctly or to identify the location and nature of the fault(s) within it. Since the numbers, types and locations of possible faults are astronomical, no amount of testing can guarantee a perfect design. Fortunately, many different faults will exhibit the same symptoms meaning that one test will detect them all. Also, some faults are more likely than others so a test for the most common faults will yield high confidence (but not certainty) in the perfection of the design. The nature and extent of testing depends, in part, on the assumptions about the design and the information desired from the test. For example, a board built from a known good design and implemented with tested parts may be presumed correct after just a few functional tests while an unproven design may require extensive testing to achieve similar confidence.

The three major types of digital logic tests are:

- Functional: does the design work?
- Characterization: are design parameters within specification?
- Troubleshooting: where are the faults? Can they be fixed?

Each type of test needs a different set of test vectors. A functional test of an adder, for example, probably concentrates on the LSB, the MSB and the sign bits with less concern for the middle bits of the adder. A troubleshooting test concentrates on each bit or logic pathway without regard for how it functions or how it interacts with other bits. A characterization test concentrates on extreme conditions such as all ones and all zeros and the parametric events during transitions between them, and on propagation time from LSB to MSB and sign. A functional test might provide a partial characterization or troubleshooting test but seldom a complete one; and similarly for the other two types.

Faults are usually classified as:

- Stuck-at faults where a node is stuck-at-one, stuck-at-zero, or stuck open.
- Bridging faults where two or more signal lines are shorted together.
- Parametric faults where one or more design parameters are outside their specifications.
- Design faults where the individual logic elements operate correctly but the ensemble, the design, does not function as intended.

Because Verilog modeling is an idealization of the logic elements, most ECE 526L testing will be directed at finding design flaws. This flaw occur when the student misunderstands the design requirements, incorrectly encodes a correctly understood requirement through typing errors or misapplication of the modeling constructs; or ignores or improperly accounts for anomalous, illegal or exception conditions. (Note that Verilog can be made to model stuck-at faults and can model specific technologies for parametric testing).

FUNDAMENTALS OF TESTING

To be detected, a fault condition must yield a different output than the fault-free design for at least one test pattern of the test suite. Every signal line should exhibit both logic states during the test. (This is a necessary but not a sufficient condition for testing that signal.) A test suite designed to detect single stuck-at faults will also detect a high percentage of multiple faults including bridging faults. Not all faults are testable due to redundancy in the design. (Redundancy means non-minimal logic but may be necessary for hazard elimination or for fault tolerance). Because it focuses on the individual nodes and signal paths in a design, stuck-at fault testing is not particularly effective at testing algorithms and functions. (Remember that the individual logic elements will probably all function correctly in a circuit with design faults).

TEST STRATEGIES

- Exhaustive test: apply all combinations of logic states to the inputs (usually as a binary counting sequence). Easy to create. Detects all detectable faults. Works only on combinational logic (but with correct signal segregation and control, can sometimes be used with sequential logic). Test time doubles for each added input. High degree of redundancy in the test suite for all but the smallest circuits and redundancy increases with number of inputs. A divide and conquer strategy can cut test time by limiting testing to independent blocks of logic.

- Random test: apply random combinations of values to the inputs. A reasonable number of random test vectors will detect a high percentage of all detectable faults. “Reasonable” may range from 10–50% of the number of exhaustive test vectors. Suffers from the same problems as an exhaustive test. Duplicate test vectors use time but detect no new faults. (Duplicate test vectors are also hard to find). No guarantee that any particular fault will be detected.

- Heuristic test: apply limited set of hand-generated test vectors. Human beings can often identify the most important areas to be tested. Is not systematic or algorithmic. Often the method of choice for functional testing. Fault detection or “fault coverage” depends on the skill of the test designer.

- Algorithmic test: use an algorithm or procedure such as the D-Algorithm, PODEM, or FAN or algebraic methods such as Boolean difference to create test patterns. Creating programs to generate the test vectors is hard but once created, test generation is easy. May not find all detectable faults. May not generate the most efficient test suite. May be the only way to generate an acceptable test suite in a reasonable period of time for large designs.

COMBINATIONAL VS. SEQUENTIAL TESTING

Combinational circuits have no elements (flip-flops, latches, memories, registers or other structures) that hold a value. Sequential circuits do contain such elements. Therefore, the tester can directly control all inputs to a combinational circuit for each test vector but lacks such direct control with a sequential circuit. Each test vector applied to a combinational circuit is independent of all other test vectors. Scrambling the test vectors changes only the order of discovery of faults. Test vectors applied to a sequential circuit, however, are not independent because they affect the memory elements of the circuit – and these are part of the test inputs for subsequent test vectors. In cases where the memory elements are themselves sequential such as shift registers and state machines, setting the desired pattern into the memory elements may require considerable effort and test time.

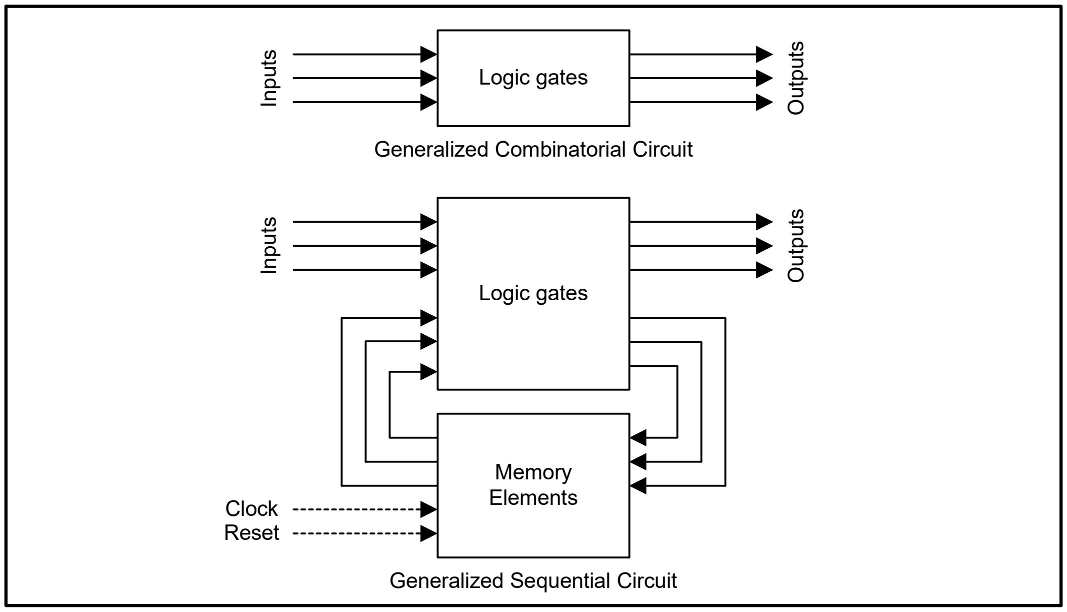

Figure 1. The difference between combinational and sequential circuits

Sequential circuit testing usually follows this general pattern: inputs are initialized and an asynchronous reset is applied (and removed) to initialize the memory elements, then various test vectors are applied to the inputs in synchronism with (but not necessarily coincident with) one edge of the clock while outputs are sampled and memory elements are updated on one edge of the clock, usually the same edge as the inputs for high speed circuits and usually the opposite edge for low speed circuits. Depending on the needs of the test, the asynchronous reset may be applied more than once. The clocking signal may be symmetric or not, may have a fixed or variable frequency and may be syncopated with stretched, shortened or missing pulses during part of the test.

FUNCTIONAL TESTING AND VERIFICATION

Functional verification, the primary testing ECE 526L students will do, must adhere to the fundamental tenet of testing: a fault, if present, must yield a different output than the fault-free case. As an example, consider a 2-input multiplexer. If zeros are applied to both inputs, then it is impossible to tell which input is actually selected. A non-zero output does indicate a fault, but if the intent was to test the selection circuit, a zero output is null, *i.e.* it gives no information. To be detected, an incorrect function must yield a different output than the correct function for at least one test pattern of the test suite.

Suppose the student sets one input of the MUX to zero, the other to one, and then toggles the select between states. Upon observing the output toggling with the select, the student concludes the model is correct… but is it? With only one state applied to either input, the output is indistinguishable from just the select (or perhaps from $\overline{\text{select}}$). To detect an incorrect control function, inputs must be toggled between states for each combination of control states. To illustrate the point further, consider a 4-input MUX. It has four control states and so requires a minimum of 8 tests (2 input states for each of the four control states) to fully verify the select function. (This also indicates how redundant an exhaustive test would be: four inputs and two controls would require $2^6 = 64$ tests).

Now imagine an 8-wide array of 2-input multiplexers. If the byte on input A is 8'h00 and the byte on input B is 8'hFF, then the inputs are distinguishable and the basic tenet is met. But what if, through mischance, the order of inputs on B is reversed. Here's another undistinguished fault. To make that fault distinguishable, additional trials with 8'h0F, 8'h33, and 8'hAA may be required (they're not the only possible patterns to distinguish this fault).

Generally, design errors reverse the ordering of a full group of signals, one set of data inputs or an entire data bus, for example, and not just a sub-group within it – provided the design does not build up the group from smaller elements or subdivide the group into separate functions. Whenever signals are grouped (*e.g.* data[15:0]) use opposite states for LSB and MSB to detect reversals of desired bit ordering. Whenever grouped signals are built up by concatenation, also test for correct bit and sub-group ordering. Suppose, for example, that a 9-bit control bus, *ctrl*, is created from a 3-bit opcode, a 2-bit mode, a direction bit (input or output) and a 3-bit address. The desired control bus is

```
[O O O M M D A A A]      control bus
 8 7 6 5 4 3 2 1 0       bit numbering
```

with the opcode in the high order bits and the other signals in order below it. To guarantee a correct control bus, the order of the sub-groups must be verified. That is, are the opcode bits in positions 8 to 6, mode bits in positions 5 and 4, and so forth. Further, is the opcode MSB in position 8, the mode MSB in position 5, and so forth.

Generally when a group of signals is subdivided with part selects, problems occur not in the module that subdivides the group but in other modules that use the sub-group. Testing must ensure that the sub-group ordering in module A is respected by module B.

Test vectors that find many faults are said to be "robust." Test time is valuable and the less of it used, the better – provided, of course, that the necessary faults are checked. Recall the statement that stuck-at fault testing was not particularly good at verifying functionality. True, but if functional test vectors are written so that stuck-at testing can also be done, the vectors are probably quite robust**. Students will be expected to write robust test vectors**, *i.e.* to minimize the number of test vectors while maximizing the number of faults checked.

Typical questions to ask (although not all questions apply to every problem):

- Are different inputs distinguishable by different patterns?
- Can reversed bit ordering be detected (especially across module boundaries)?
- Can shorts between adjacent bus signals be detected?
- Do the asynchronous functions work: does SET make a low output go high and a RESET make a high output go low? Does ENABLE work? Does a tri-state ENABLE work?
- Do the synchronous functions work: do outputs change only on the right clock edge?
- Do the asynchronous functions override the synchronous functions?
- Is the state table or transition table verified with the given test suite? Have unused states been checked?
- Is the control table or are the control functions verified with the given test suite? Have data inputs been toggled for every combination of control states? Have unused combinations or states been checked?
- Are anomalous, illegal or exception conditions tested and does the circuit conform to specification or at least do something reasonable. Does the model generate a warning message for illegal inputs?
- Have all inputs and outputs and as many internal signal lines as possible been switched between logic states?
- Have undefined ('b**x**) and high impedance ('b**z**) states been applied to critical inputs?
- Have the input and output extremes been tested (largest, smallest, positive, negative, leftmost, rightmost, etc.)
- Has the test suite been documented to indicate what is tested, assumptions made, and tests omitted (if any)?

COURSE REQUIREMENTS AND CAUTION

Testbenches must include a test strategy and must, as a minimum, ensure that all signals to and from the module(s) under test change state at least once during the test. Testing should show an understanding of the functions being tested. An exhaustive combinational pattern should not be blindly applied to sequential logic, for example. Signals should not be undefined, floating, or un-initialized. Violations will be penalized.

Behavioral Coding Style and Errors

Most labs will be written in behavioral Verilog/SystemVerilog, though the first ones will use structural modeling. In behavioral code, there are several techniques that may simulate correctly but would either fail to produce any hardware at all or produce defective hardware when synthesized. These stylistic errors are easy to avoid. While the list below is not exhaustive in that there is an infinite variety of ways to produced defective designs, by following these guidelines many common errors can be avoided.

1. Always have an “else” clause for every “if.”
2. Have a default clause for every case statement.
3. If one signal in a sensitivity list is edge sensitive, all must be.
4. Avoid all use of combinational feedback. Sequential feedback is fine.
5. Avoid all use of continuous assignments for anything other than bidirectional port interfaces.
6. Do not put explicit time delays into circuit descriptions. Their use is limited to non-synthesizable test fixtures.
7. Clock and reset signals are always primary inputs.
8. While there are sometimes valid reasons for violating synchronous design guidelines, none of them apply to the labs in this manual. Synchronous design is mandatory for all designs in this course.
9. Never split assignments to any one variable between “always” blocks.
10. Use only one assignment operator (blocking or non-blocking) for any operand.

Some of the above rules are likely to seem esoteric and incomprehensible at the start of the semester. Refer back to this list as the semester progresses and you learn more about behavioral hardware description.