To appear in *Proceedings International Conference on Microelectronics Systems Education*, June 2011

HDL Coding Guidelines for Student Projects

John A. Nestor
Department of Electrical and Computer Engineering
Lafayette College
Easton, Pennsylvania
nestorj@lafayette.edu

***Abstract*— Hardware Description Languages and FPGAs make it possible for students to design large, complex circuits. However, HDL-based design is fraught with pitfalls that students often find difficult to understand and avoid. This paper describes a set of design guidelines for HDL-based design taken from a variety of sources and distilled into a concise list. These guidelines are intended to help students avoid these pitfalls and create reliable designs. They have been used in a Senior Design Project course in which students design a wireless network.**

Keywords-Hardware Description Languages; HDL-Based Design; FPGA Design; Undergraduate Education

I. INTRODUCTION

Inexpensive FPGA boards and low-cost or free design tools make it possible for students to execute design projects of unprecedented complexity using Hardware Description Languages (HDLs). At the same time, HDL-based design is an important skill that is attractive to many prospective employers.

However, a number of issues make it difficult for students to effectively learn HDL-based Register-Transfer Level (RTL) design and to create and debug complex projects. First of all, the languages themselves have flaws. Both VHDL and Verilog were initially designed as languages for event-driven simulation, and only a subset of the language can be synthesized into hardware. Code in this subset must be carefully written to avoid common pitfalls such as latch inferences. Second, and perhaps more importantly, students must escape the common conceptual error of equating HDL code with procedural software and think instead in terms of hardware. Finally, students must learn the discipline of thoroughly verifying their HDL code using testbenches [1].

To address these issues, students need a concise set of guidelines for HDL-design. While existing guidelines do exist (e.g., [2-8]), they are often too lengthy for students to absorb and use effectively. At the same time, these guidelines may not address common student pitfalls.

This paper describes a set of concise but comprehensive guidelines for students doing HDL-based design using the Verilog HDL. These guidelines have been drawn from a number of different sources and distilled into a short and concise document. The guidelines were chosen based on several years of experience observing the pitfalls encountered by students as they attempt to design and debug complex FPGA designs in a senior design project at Lafayette College [9].

These guidelines are based on a set of coding *templates* – coding patterns that illustrate the underlying hardware of coding constructs for combinational and sequential logic.

The guidelines are organized so that they go from very general guidelines that encourage students to think of the underlying hardware to very specific guidelines that help students avoid common pitfalls. While these guidelines are for Verilog, a similar version could be developed for VHDL.

The remainder of this paper is organized as follows. Section II describes the coding templates that are central to the guidelines, while Section III discusses each of the six categories and summarizes the motivation for important individual guidelines. Section IV describes our experience using these guidelines with students in the lab. Section V concludes the paper and suggests future work. The Appendix contains the complete guidelines.

II. CODING TEMPLATES FOR SYNTHESIS

The coding guidelines are based on a set of *templates* that show how fragments of synthesizable Verilog are translated into hardware as shown in Figure 1. These templates are used to teach HDL-based design at Lafayette in courses ranging from sophomore digital circuits classes to senior project and elective courses [9].

Combinational logic elements (a) consist of a set of logic gates with outputs that connect directly to the outputs of the element. These can be described using the `assign` statement (e.g., `assign sum = a ^ b ^ c;`) or an `always` construct that describes the output as a function of its inputs using a sequence of procedural statements.

Registered logic elements (b) consist of logic gates and flip-flops, with all outputs passing through flip-flops before connecting to other logic elements. They can be described Verilog using the `always @(posedge clk)` construct along with a sequence of procedural statements that specifies the values that outputs take on after a clock edge.

More complex circuits such as state machines must be constructed with a combination of combinational and registered `always` blocks.

Students are also provided a template for *self-checking testbenches* – non-synthesizable blocks that are used in simulation to generate input stimulus patterns for a module under verificaton while monitoring the module's response for correct operation [1].

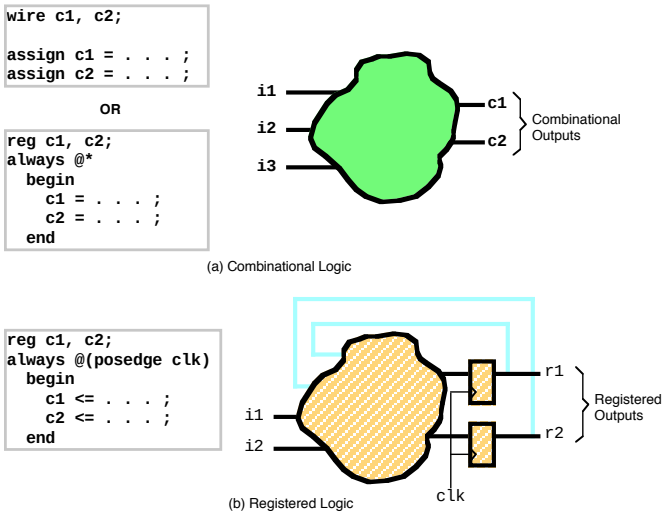

Figure 1. Synthesizable Logic Elements

III. Overview of the Guidelines

This section summarizes the design guidelines, which are organized in seven categories: General Guidelines, Basic Formatting, Coding Guidelines, Combinational Logic, Registered and Sequential Logic, Verification, and Clocks and Timing. They can be examined in detail in the appendix.

A. General Priniciples for Success

The first section of the guidelines is intended to describe general principles for successful design. Most important of these is to understand the underlying hardware that is described in the previous section. Other general principles include learning to avoid common design pitfalls, writing and using testbenches to verify designs thoroughly through simulation, and being systematic in debugging. While these may seem obvious, many of the problems we have observed in student projects have resulted directly from their failing to follow these guidelines and, in particular, attempting to write RTL code using a procedural "software" model.

B. Basic Formatting

This section provides guidelines for formatting code, naming variables and I/O ports, using symbolic constants, and using consistent ordering in multi-bit signals. Following these guidelines will result in the creation of code that is easer to read, understand, and maintain.

C. Combinational Logic

This section provides guidelines for how to describe combinational logic. These include the use of the continuous assignment statement for simple expressions, use of the `always @*` construct for complex descriptions, ways to avoid errors resulting from mixing multiple-bit and single-bit expressions, and proper use of the `case` statement for complex behavior. The guidelines also discourage code that is legal Verilog but either results in ambiguous, error-prone specifications and code that "works in simulation" but not when synthesized into hardware.

D. Registered and Sequential Logic

This section provides guidelines for working with sequential logic, including registered logic synthesized from "clocked" always blocks, sequential logic that combines registered and combinational logic, and state machines. It includes guidelines to avoid common pitfalls such as using blocking assignments rather than non-blocking assignments and failing to specify enough bits in state variables.

E. Designing with Hierarchy

This section provides guidelines for creating a hierarchical design using module instantiation. It includes guidelines for avoiding common errors when using module instantiation (e.g., misaligned connections) and for creating a cleanly partitioned hierarchy.

F. Clocks and Timing

This section provides guidelines for synchronization of asynchronous inputs and the reliable design of single and multiple-clock sequential circuits Guidelines include using a single clock with the same active edge when possible, synchronizing signals that pass between clock domains when a single clock is not possible, avoiding clock gating, and using the proper level of handshaking to reliably pass control and data between concurrently operating state machines [8].

IV. Preliminary Results

Although we have used coding guidelines in some form for the last several years, the guidelines described in this paper were used for the first time during the Fall 2010 semester in a required Senior Design Project at Lafayette College. Students in this project design a wireless network interface that is implemented using an FPGA board with a small peripheral board for the RF front end [9]. Because it is a required course students begin this project with a range of abilities, motivation, and affinity for digital design.

Students work in teams of 2-3 and independently design the network interfaces which are tested for interoperability with each other. During Fall 2010 there were four teams. All four teams were able to complete the basic functional requirement to send and receive packets, while three teams implemented advanced features such as CRC error checking and the ability to request and receive acknowledge packets in return. This level of success was better than encountered in previous offerings of the course, and we believe that these guidelines were a factor in this success.

V. Conclusion

This paper has described a set of guidelines for students using Verilog HDL to design complex circuits implemented in FPGAs. Although hard assessment data is not available, it appears that these guidelines have improved students' ability to successfully complete design projects. Future work will include the use of these guidelines in lower-level digital design and computer organization courses and the development of a textbook manuscript that provides a comprehensive treatment of RTL design using these guidelines.

APPENDIX – COMPLETE GUIDELINES

1. General Priniciples for Success

1.1 When writing Verilog code, think in terms of hardware, not software. Understand and use the templates for combinational logic, registered logic, sequential logic, and finite state machines.

1.2 Be mindful of common design pitfalls and follow these guidelines to avoid them.

1.3 Write testbenches and thoroughly simulate Verilog modules before attempting to debug in hardware.

1.4 Read, understand, and address error messages and warnings in both the simulator and synthesis tool

1.5 Be systematic in debugging. Isolate and address problems one at a time.

2. Basic Formatting

2.1 Create a separate Verilog file for each module. Name each file `<module_name>.v`.

2.2 Include a header comment block in each Verilog file which lists the module name, a brief description of the module function, author name(s), creation date, version, and revision history.

2.3 Limit line lengths to 100 characters or less.

2.4 Use a consistent indentation style that indents 2 spaces at each level. Avoid using tab characters in indentation.

2.5 Include comments in your code to improve readability and understandability when the meaning of the code itself is not obvious.

2.6 Use lowercase and underscore characters for module, port, signal, and variable names, e.g. `adder`, `rdy`, `int_priority`, etc.

2.7 Use uppercase and underscore characters for parameter, symbolic constant, and module instance names, e.g., `WIDTH`, `RED_ALERT`, `EOT`, `U_ADDER`, etc.

2.8 Use a consistent set of names for signals that are used in multiple modules, e.g., `clk`, `rst`, etc.

2.9 Use the suffix "`_n`" on signal names that are active low, e.g., `rst_n`.

2.10 Use a consistent ordering for the indexes of multiple-bit vectors `[high:low]` to avoid accidentally reversing bits. For example:

```
wire [7:0] a, b;
```

2.11 Use symbolic constants in your code instead of "magic numbers". Use the `parameter` construct for local constants. Use the `` `define `` compiler directive for global constants (parameters are local to a module).

2.12 Collect global symbolic constant definitions into a header file and use the `` `include `` directive to include these declarations in source files that use them.

2.13 Use a consistent order for module port declarations:
Inputs: clocks, resets, enables, control signals, data
Outputs: clock, resets, enables, control signals, data
Bidirectional signals.

2.14 Indent declarations so that port names and signal names are aligned.

3. Combinational Logic

3.1 Use continuous assignment statements (`assign`) to describe combinational logic when the circuit's behavior can be described using simple expressions.

3.2 Use the power of Verilog's high-level operators to write functional descriptions rather than low-level descriptions. Avoid cryptic and overly clever code. For example, use:

```
assign cmp_out = (in1 == in2);
```

instead of

```
assign cmp_out = ~|(in1 ^ in2);
```

3.3 Use logical Boolean operators (e.g., `&&`, `||`, `!`) when computing a single Boolean true/false value. Use bitwise Boolean operators (e.g., `&`, `|`, `~`) when computing a multiple-bit value. Never mix them.

3.4 When using `always` blocks for combinational logic:

a. Use the Verilog 2001 construct `always @*` when supported.

b. Otherwise, make sure to include all inputs in the sensitivity list e.g.:

```
@always(in1 or in2 or in3)
```

c. Remember that values assigned in procedural statements are variables and must be declared as `reg`s.

d. Assign outputs using blocking assignments (=) [4].

e. Make sure that a variable is assigned values by exactly one `always` block

f. Never use `assign` or `deassign` within an `always` block.

g. Don't use delay (#) operators in RTL code.

3.5 Use an expression that evaluates to a single bit true/false value as the condition for an `if` statement. Don't use multiple bit values as the condition for an `if` statement

3.6 To avoid a latch inference in an `if` statement that assigns a variable, either include a corresponding `else` statement that assigns the same variable or include an assignment statement prior to the if statement.

3.7 To avoid a latch inference in a `case` statement that assigns a variable in one case item either (a) include an assignment to the variable in all case items including the default item; or (b) include a "default value" assignment statement prior to the case statement.

3.8 Always use `default` in `case` statements unless a prior assignment sets a default value or the case items explicitly cover "`x`" and "`z`" values.

3.9 Use `casez` to specify item values with "wild card" digits. Express "wild card" digits using '`?`' instead of '`z`'. Do not use the `casex` statement.

3.11 Do not use the "`full_case`" or "`parallel_case`" synthesis directives.

3.12 When implementation cost is not a concern, assign default output values to "`0`" to avoid simulation/synthesis mismatches. When cost is a concern, assign default output values to "`x`" to indicate a don't care condition [7].

3.13 NEVER allow cycles in combinational logic connections. This includes cycles created by connections through more than one block!

4. Registered and Sequential Logic

4.1 Use flip-flops for sequential logic. Never use latches.

4.2 Assign registered outputs using nonblocking (`<=`) assignments [4].

4.3 Use asynchronous reset only for system reset and initialization. NEVER tie an asynchronous reset to the output of a combinational logic function.

4.4 Place state machine code in a module to limit visibility of its internals and avoid conflicts with other code.

4.5 Use one of the following recommended styles for coding a state machine [5]:

a) One clocked `always` block for the state register and one combinational `always` block for the next state and output logic (Both Moore and Mealy machines).

b) One clocked `always` block for the state register, one combinational always block for the next state logic, and one clocked `always` block for registered outputs (Moore machines only).

4.6 Make sure that state registers are sized to accommodate the number of states in the state diagram.

4.7 Include a reset in the state register.

4.8 Define FSM state names as sized symbolic constants using `parameter`. Assign the parameter a size to avoid errors.

4.9 Use meaningful names for state names (e.g., `IDLE`, `START` instead of `S0`, `S1`).

4.10 Assign next state and all outputs to default values prior to the `case` statement for a state machine.

4.11 Don't assign a default next state value using the unknown ("`'bx`") value [7]. Instead assign an error state or initialization state to allow the state machine to recover from an unknown state or condition if it occurs in hardware.

5. Designing with Hierarchy

5.1 Use module instantiation to implement your module hierarchy with a top-level module that has a module name (and file name) in all capitals.

5.2 When instantiating a module, use an instance name that is all capitals and begins with the prefix "`U_`" (e.g. `U_COUNT1`).

5.3 Use the explicit connection style when instantiating modules with more than 3 ports.
Example:

```
counter U_COUNT1 (.clk(clk1),
 .rst(rst),.enb(cnt_enb). .count_r(q));
```

5.4 Don't use concatenations when connecting module ports. If a concatenation is necessary, use a separate `assign` statement.

5.5 Avoid mixing structural and behavioral code in a module – a module should usually contain either all structural code (i.e. module instantiations) or all behavioral code (i.e. always blocks, assign statements, etc.)

5.6 At each level of hierarchy, design clean interfaces between modules that minimize the number of connections.

5.7 When possible, use registered module outputs to ease timing analysis.

5.8 Avoid glue logic in the top levels of your design hierarchy.

6. Clocks and Timing

6.1 (Optional) Use registered outputs to improve performance and speed up timing analysis.

6.2 Use a common clock edge for to trigger all flip-flops in your design. Don't mix clock edges. Use the positive edge only unless absolutely necessary.

6.3 Synchronize all asynchronous inputs using a flip-flop as a synchronizer.

6.4 Don't use multiple synchronizers for the same input.

6.5 When synchronizer failure is a concern, use two flip-flops in series to make a dual-stage synchronizer and increase available resolution time.

6.6 If possible, use a single clock for your entire system. If that isn't possible, synchronize signals that pass from one clock domain to another.

6.7 Don't use gated clocks.

6.8 Use handshaking to reliably transfer data between clock domains.

6.9 When performance is a concern, consider using a FIFO to transfer data between clock domains.

REFERENCES

[1] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*, Springer, 2003.

[2] M. Keating and P. Bricaud, *Reuse Methodology Manual for System-on-a-Chip Designs*, 3rd ed., Springer, 2007.

[3] Freescale Corporation, "Verilog HDL Coding Semiconductor Research Standard", Version 3.2, February 2005.

[4] C. Cummings, "Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill", *Proceedings SNUG 2000*.

[5] C. Cummings, "Synthesizable Finite State Machine Design Techniques Using the New SystemVerilog 3.0 Enhancements", *Proc. SNUG 2003*.

[6] C. Cummings, "full_case parallel_case, the Evil Twins of Verilog Synthesis", *Proc. SNUG Boston Meeting, 1999.*

[7] M. Turpin, "The Dangers of Living with an X (bugs hidden in your Verilog)", Research Paper, *Proc. SNUG Boston Meeting*, Oct. 2003

[8] C. Cummings, "Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs", *Proc. SNUG San Diego Meeting*, 2001.

[9] J. A. Nestor, J. Greco, and I. Jouny, "HDL-Based Instruction across the ECE Curricula", *Proceedings of the IEEE Frontiers in Education Conference*, October 2010.