Verilog Circuit Rules

Greg Gibeling
UC Berkeley
gdgib@eecs.berkeley.edu

November 1, 2007

1 Purpose

This document is a comprehensive list of which Verilog constructs are allowed in CS61C. You may wish to print it out to have handy, but if you do so, keep an eye out for updates, as mistakes are found. Do not worry if you don't know these constructs, we have not covered them all, as they are prohibited anyway.

2 SubSets of Verilog

Structural: Structural Verilog modules consist only of **input**, **output** and **wire** declarations with primitive gates or other module instantiations. Structural Verilog is the subset closest to true circuit implementations, and will almost always be an acceptable implementation methodology in this class.

Continuous: Continuous Verilog modules may include anything from the Structural Verilog subset, with the addition of **assign** statements. **assign** statements provide for continuous (always happening) assignment, and are a good way to transcribe boolean equations (& for AND, | for OR, ~ for NOT). We will often allow you to use Continuous Verilog, but some assignments in CS61C will restrict the you to using 1bit values in your **assign** statements.

Procedural: Procedural Verilog is the largest subset of the language, and includes all of the above, plus **always** and **initial** along with the signal type **reg** which they require. You will never be allowed to use Procedural Verilog for anything except testbenches. Note that many of the modules we will give you use Procedural Verilog. We are in effect cheating on the rules by doing this. If you want to learn to use Procedural Verilog safely, and be allowed to use it in circuits, you take EECS150.

3 Restrictions

Always blocks: **always** blocks may be used in testbenches, or to build registers. **always** blocks for registers must be of the form **always** @ (**posedge** Clock) and must use nonblocking assignment <=. You should generalize registers from our examples, rather than write your own. You may not put any combination logic in a register, in other words an **always** @ (**posedge** Clock) block may not contain even a simple counter: the increment (add one) logic must be implemented outside of the **always** block.

Variable Index: Variable-indexed shifts (<< or >>) and bit-selects (x[y] or x[y:z]) are not allowed. Variable-indexed register files are okay, but we will give you code for these. This restriction means that you can use shifts and bit-selects, but that the indices must be constants.

Looping: The keywords **forever**, **repeat**, **while**, **for**, **fork** and **join** may appear only in testbenches.

Macros: The keyword **'timescale** must appear in every testbench, and nowhere else.

4 Prohibitions

Non-Circuits: These are constructs which are difficult if not impossible to translate into circuits. As a result you may not use them, except in testbenches.

Directives: Any statement beginning with a $, such as **$display**.

Drive strength: Anything relating to drive strength.

Undrived signals: The constant 1'bz or the tests for it === or ==\!

Nesting: The keyword **assign** may not be nested with an **always** or **initial** block.

Keywords: The keywords **force**, **release**, **deassign**, **disable**, **parameter** and **localparam**.

Operators: Multiplication (∗), division (/) and modulus (%).

Events: The keywords **wait**, @event and **specify**.

Tasks: The keywords **function**, **task** and **generate**.

Non-Bit types: **integer**, **signed**, string, **real**, **time** and **realtime**.

Macros: **‘define**, ‘undef, **‘ifdef**, **‘else**, **‘endif**, **‘default_nettype**, **‘include**, **‘resetall**, **‘unconnected_drive** and **‘celldefine**.

Non-standard signal types: **tri**, **wor**, **trior**, **wand**, **triand**, **trireg**, **tri1**, **tri0**, **supply0** and **supply1**.

UDP's: User defined primitives.