Use of Verilog in CS/ECE 552
Adapted from Andy Phelps CS552-Sp'06 document
Last Updated: February 4, 2008

Table of Contents

1. Introduction
2. Justification of Limitation
3. Allowed Keywords
4. Allowed Operators
5. Usage Examples
6. CS 552 Verilog Check Program

1. Introduction

In the CS 552 course you will be limited to a small subset of the complete Verilog language. The purpose of this document is to outline and explain these usage restrictions. It is not a complete Verilog reference. You can consult the following resources for a more complete documentation of the language:

- On-line Verilog HDL Quick Reference Guide
- ASIC World Verilog Quick Reference

2. Justification of Limitation

The intention of limiting the Verilog language is to force you to "think like hardware". Verilog in its full form is a powerful HDL that can very nearly be used as a programming language with capabilities similar to C. While this is useful for experienced designers trying to increase productivity, there is no intellectual benefit to be gained by using the abstract features of Verilog. Instead, you will write Verilog in a manner that reflects the actual hardware you are trying to design. Think of it as verbally describing the components of a schematic sheet.

In addition to the educational value of using a subset of the Verilog specification, limiting your code to only well-understood constructs will also help to avoid some pitfalls which can lead to hard to find bugs. Verilog designs have been found which simulate correctly or incorrectly, basically at random, depending on what order certain assignments happen to execute in. Designing with a limited subset of Verilog will tend to produce good designs which synthesize well; for this reason you may be required to follow similar rules when you have a job in industry.

3. Allowed Keywords

Verilog keywords in this course are grouped into three categories: always allowed, allowed with stipulations, and never allowed. These groups can be found in Table 1. This list is not guaranteed to be complete. If you are ever in doubt about the use of a Verilog keyword consult the TA before proceeding.

<table>
<thead>
<tr>
<th>Always Allowed</th>
<th>Allowed with Stipulations</th>
<th>Never Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>assign</td>
<td>case</td>
<td>attribute</td>
</tr>
<tr>
<td>module</td>
<td>casex</td>
<td>pulldown</td>
</tr>
<tr>
<td>endmodule</td>
<td>reg</td>
<td>pullup</td>
</tr>
<tr>
<td>input</td>
<td>always</td>
<td>rcmos</td>
</tr>
</tbody>
</table>

Verilog Keyword Summary
Keywords in the **allowed with stipulations** group can only be used in conjunction with a case statement. The following stipulations apply:

- **case, casex** - the default clause is required. If the default clause is used during execution, an error must be asserted. Large case statements (~12 lines or more) must go in their own module.
- **reg** - can only be used to specify the outputs of a case statement.
- **always** - can only be used to introduce a case statement.
begin - can only be used to introduce a case statement.
end - can only be used to terminate a case statement.

4. Allowed Operators

In this course you will be limited to using simple boolean logic operators. You may use the shift operators, but only by a constant amount (i.e. sigY << sigX is not allowed, but sigY << 4 is). The ternary operator (a? b:c) is allowed and even encouraged. Concatination ( {bit, bit} ) and reduction ( &a[31:0] ) are allowed. Logical equality (==) and inequality (!=) operators are also permitted.

You are expressly forbidden from using arithmetic operators (+, -, *, /, %) and less-than or greater-than comparisons.

Table 2 summaries the use of Verilog operators

<table>
<thead>
<tr>
<th>Allowed</th>
<th>Not Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>~m</td>
<td>Inversion</td>
</tr>
<tr>
<td>m &amp; n</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>m</td>
<td>n</td>
</tr>
<tr>
<td>m ^ n</td>
<td>Bitwise XOR</td>
</tr>
<tr>
<td>m ~^ n</td>
<td>Bitwise NXOR</td>
</tr>
<tr>
<td>&amp;m</td>
<td>AND Reduction</td>
</tr>
<tr>
<td>~&amp;m</td>
<td>NAND Reduction</td>
</tr>
<tr>
<td></td>
<td>m</td>
</tr>
<tr>
<td>~</td>
<td>m</td>
</tr>
<tr>
<td>^m</td>
<td>XOR Reduction</td>
</tr>
<tr>
<td>~^m</td>
<td>NXOR Reduction</td>
</tr>
<tr>
<td>m == n</td>
<td>Equality</td>
</tr>
<tr>
<td>m != n</td>
<td>Inequality</td>
</tr>
<tr>
<td>m === n</td>
<td>Identity</td>
</tr>
<tr>
<td>m !== n</td>
<td>Not Identical</td>
</tr>
<tr>
<td>m &lt;&lt; const</td>
<td>Shift Left*</td>
</tr>
<tr>
<td>m &gt;&gt; const</td>
<td>Shift Right*</td>
</tr>
<tr>
<td>test ? m:n</td>
<td>Ternary</td>
</tr>
<tr>
<td>{m, n}</td>
<td>Concatenation</td>
</tr>
<tr>
<td>{m{n}}</td>
<td>Replication</td>
</tr>
</tbody>
</table>

5. Usage Examples

The most fundamental thing to keep in mind is that your hardware design will be composed of two things: combinational logic, and state. For any moderately complex logic design, state must be handled in an organized way. An R-S flipflop would get you laughed out the door of a computer design company! In our designs, state will all be held in D-flipflops that are all clocked on the rising edge of a common system clock. You are provided here with a module for a single D-flipflop with synchronous reset. Instantiate copies of this module to build larger registers. Do not create your own flipflops in some other way.

Combinational logic will be specified primarily with "assign" statements. The statement "assign xyz = a & b;" specifies an AND gate. An assign statements can get long and complex, and can specify hundreds of gates,
but it always represents some set of flow-through logic that generates the value for a wire (or bundle of wires).

Your design will be hierarchical. That means you will write Verilog modules for all the small building blocks of your design, and you will instantiate these modules within larger modules. For example, if you want to increment a number, do not create new increment logic each time; write increment modules of whatever sizes are needed and call them from your other modules. Most of your design should consist of calls to other modules.

The modules you write should have a structure like this:

```verilog
module my_module_name (param, param, ... param);
    input param;
    output param;
    wire [msb:lsb] signalName;
    assign signalName = expression;
    modulename instance (param, param...); // call some submodule
endmodule
```

First, we name the module and list all its input and output parameters (in order). Second, we list all the input and output parameters all over again, giving their sizes, e.g.

```verilog
    input [7:0] firstByte;
```

Put the input and output statements in the same order that the parameters appear in the module statement. Verilog does not require it, but good practice (and this class) require that you do so.

The assign statements specify all logic that is to be done in this module. Care should be taken to format it so that it is readable; use liberal whitespace and consider lining up similar logic into columns.

Note that assign statements can operate on an entire "vector" of bits at one time. Sometimes this involves making multiple copies of a 1-bit wire in order to interact with the vectors. For example:

```verilog
    wire [15:0] a, b, ss, ans;
    assign ss = {16{s}}; // use 16 copies of "s"
    assign ans = (ss ^ a) | b;
```

When instantiating a module, you must use name the ports when connecting wires to it. For example:

```verilog
mux2_1 m0(d0, d1, s, b) is NOT OK
```

```verilog
mux2_1 m0(.input0(d0), .input1(d1), .select(s), .output0(b)) is CORRECT
```

Although a case statement is a fairly high-level construct, it is an extremely useful way of representing muxes, next-state evaluation, and other common types of logic. The case statement will be the only situation in this class where you will use statements such as begin, end, reg, always. Look at this example:

```verilog
    wire [1:0] s;
    reg out;
    always @* case (s)
        2'b00 : out = i0;
        2'b01 : out = i1;
        2'b10 : out = i2;
        2'b11 : out = i3;
    endcase
```

The value for the outputs of the case statement must be specified in every case. This is important: Failure to specify a value for some output bit will cause it to retain its previous state, causing a glitch-prone RS latch to form.
Here is a slightly less trivial example. This case statement implements the logic for a state machine which looks for pulses on inputA and then inputB. Note that all outputs are set in each case. Note also that it is clear by inspection that all combinations of inputs have been considered.

wire inputA, inputB, goBack;
wire [2:0] currentState;
reg [2:0] newState;
reg out, err;

always @(goBack or currentState or inputA or inputB)
case #1 (goBack, currentState, inputA, inputB)
  6'b1_??_??_?? : begin out = 0; newState = 3'b000; err = 0; end
  6'b0_000_0_0 : begin out = 0; newState = 3'b000; err = 0; end
  6'b0_000_1_0 : begin out = 0; newState = 3'b001; err = 0; end
  6'b0_001_0_0 : begin out = 0; newState = 3'b010; err = 0; end
  6'b0_001_0_1 : begin out = 0; newState = 3'b011; err = 0; end
  6'b0_010_0_0 : begin out = 0; newState = 3'b010; err = 0; end
  6'b0_010_1_0 : begin out = 0; newState = 3'b011; err = 0; end
  6'b0_010_1_1 : begin out = 0; newState = 3'b100; err = 0; end
  6'b0_011_0_0 : begin out = 0; newState = 3'b011; err = 0; end
  6'b0_011_0_1 : begin out = 0; newState = 3'b100; err = 0; end
  6'b0_100_??_?? : begin out = 1; newState = 3'b000; err = 0; end
  6'b0_101_??_?? : begin out = 0; newState = 3'b000; err = 1; end
  6'b0_110_??_?? : begin out = 0; newState = 3'b000; err = 1; end
  6'b0_111_??_?? : begin out = 0; newState = 3'b000; err = 1; end
  default : begin out = 0; newState = 3'b000; err = 1; end
endcase

Also not allowed: Any notion of time or delays; any real numbers. Force statements are OK for debugging but must not appear in a finished assignment.

6. CS 552 Verilog Check Program

A Java program Vcheck will be used to scan your design for some of the common illegal constructs. It is fairly simple, and can be easily fooled into either allowing things or complaining incorrectly. But it is a useful tool if you are unclear as to whether or not your design meets the requirements set forth here. You will be required to run it on your Verilog designs and hand in the output; To run it, copy the two class files Vcheck.class and VerFile.class into a directory, and from that directory type "java Vcheck <myfile.v>"

Method 2: From the unix prompt on a CS machine, cd to the directory where your verilog files are and issue the following command: vcheck.sh <myfile.v> <myfile.vcheck> Method 3: To run vcheck on all the verilog files in a directory: vcheck-all.sh