EECS 270 Verilog Reference: Combinational Logic

1 Introduction

The goal of this document is to teach you about Verilog and show you the aspects of this language you will need in the 270 lab. Verilog is a hardware description language—rather than drawing a gate-level schematic of a circuit, you can describe its operation in Verilog. Its structure is very similar to C in many ways—the same style of comments, the same operators, similar control structures, and so on. However, Verilog has one major difference from C (and any other programming language): its execution is inherently parallel, which means that events will not happen sequentially, as you are used to seeing, but at the same time.

2 Syntax and organization

To help explain the main features of Verilog, let us look at an example, a two-bit adder built from a half adder and a full adder. The schematics for this circuit are shown below:

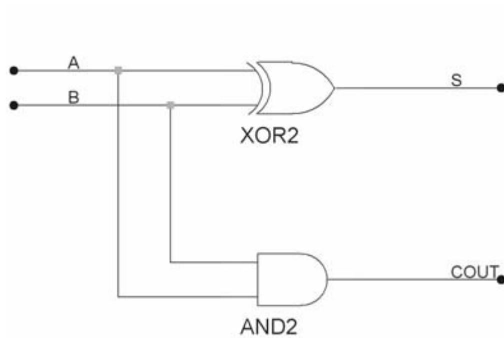

Figure 1a: Half adder

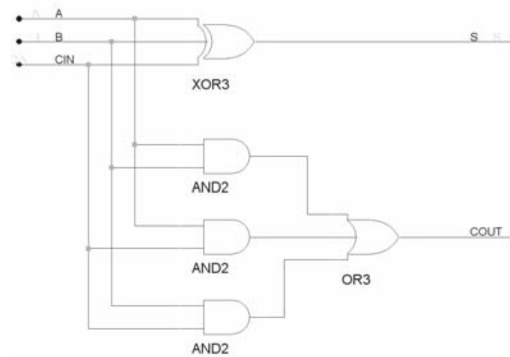

Figure 1b: Full adder

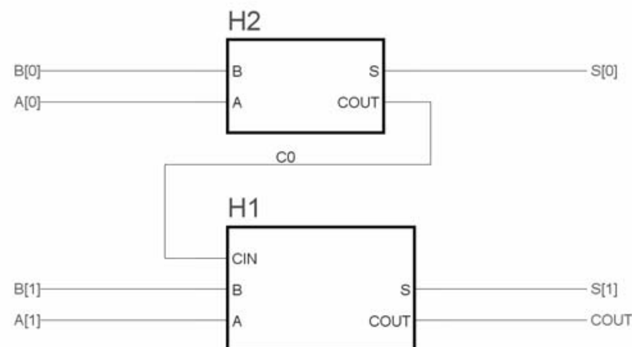

Figure 2c: Two-bit adder built from half adder and full adder

To implement these same circuits in Verilog, we can write the following code:

```
module add_half (a, b, s, cout);

    input  a, b;
    output s, cout;
    wire s, cout;

    assign s = a ^ b;
    assign cout = a & b;

endmodule    // end of half adder module

module add_full (a, b, cin, s, cout);

    input a, b, cin;
    output s, cout;
    wire s, cout;

    assign s = a ^ b ^ cin;
    assign cout = (a & b) | (a & cin) | (b & cin);

endmodule    // end of full adder module

module add_2bit (a, b, s, cout);

    input [1:0] a, b;   // Both a and b are 2 bit inputs
    output [1:0] s;     // s[1] = MSB of s, s[0] = LSB of s
    output cout;
    wire [1:0] s;
    wire cout;
    wire c0;            // intermediate carry between adders

    add_half a1(a[0], b[0], s[0], c0);
    add_full a2(a[1], b[1], c0, s[1], cout);

endmodule
```

2.1 Modules, Inputs, and Outputs

The basic organizing unit in Verilog is the *module*, a “black box” for which the internals are invisible to the outside world. Every module starts with a line of the form:

```
module <module name> (<input list>, <output list>);
```

where <input list> and <output list> are comma-separated lists of identifiers. You must declare these identifiers as inputs and outputs in the first lines of the module, as shown in the example. The default size for all signals is one bit, but you can declare them to be larger, as you can see in the add_2bit module—a, b, and s are all 2-bit buses.

You must also declare a data type for all outputs. Verilog has two data types—wire and reg. A wire cannot hold state and is always evaluated in terms of other values. A reg (short for register) will hold the last value assigned to it until another assignment changes its value. We will discuss registers in more detail in the sequential logic overview, as you will not use them before Lab 6.

You can use modules inside of one another, as shown above—the add_2bit module is built using an add_half module and an add_full module. All module instantiations take the form:

```
<module name> <instance name> (<parameter list>);
```

The parameter list can take two forms. The first is shown in the example above and is similar to parameter lists for C function calls—the inputs and outputs must be in the exact same order as in the module declaration itself. The other way to list parameters for a module instantiation is to explicitly assign signals to the correct ports, in which case the order does not matter. The instantiations from the add_2bit module are rewritten below using this form.

```
add_half a1(.a(a[0]), .b(b[0]), .s(s[0]), .cout(c0));
add_full a2(.a(a[1]), .b(b[1]), .cin(c0), .s(s[1]), .cout(cout));
```

2.2 Intermediate values

In addition to inputs and outputs, you can declare intermediate signals, which are similar to variables in a C function in that they can help you break a complex circuit into manageable parts. They can also be used to handle macro inputs or outputs that are not inputs or outputs for the module as a whole; the c0 signal in the add_2bit module is an example of such a usage. In combinational logic, all intermediate signals are of type wire.

2.3 Literals

You can define constant values (*literals*) of the form:

<size><base format><number>

where <size> contains decimal digits that specify the size of the constant in the number of bits. Although the <size> is optional, it is always a good idea to specify it. The <base format> is the single character ' (a single quote, found on the same key as the double quote (“)) followed by one of the characters b, d, o and h, which stand for binary,

decimal, octal and hexadecimal, respectively. The <number> part contains digits which are legal for the <base format>. Some examples:

```
549    // decimal number
'h8FF  // hex number
'o765  // octal number
4'b11  // 4-bit binary number 0011
5'd3   // 5-bit decimal number
```

Macros can be used to give descriptive names to literal values to make your Verilog easier to read; all macros use the ` character (left tick; on the same key as the tilde (~)). Macros can be used any place a literal can be used; when you do use them, you must place a left tick before the macro name. An example is below.

```
`define CONST3 3'b011
a = b & `CONST3;
```

2.4 Operators

The logical operators available to you in C are available in Verilog and are listed below, along with some other useful operators. Although Verilog has the standard arithmetic operators (+, -, *) as well, we prefer that you do not use them and implement everything using logical operations.

&	Bitwise AND
|	Bitwise OR
~	Bitwise negation (can generally be combined with another operator, so ~& is bitwise NAND)
^	Bitwise XOR
<<	Left shift
>>	Right shift
{}	Concatenation {a,b,c} puts a, b, and c after one another into a single value {n{m}} makes a single value that is n copies of m, one after the other

The conditional operator is also particularly useful; it assigns one of two values depending on the conditional expression. For example, if you are designing a 1-bit multiplexer with data inputs in0 and in1 and select input sel, you can write the output assignment using this operator as follows:

```
assign mux_out = (sel == 1) ? in1 : in0;
```

Note that, as in C, you could write the condition (sel == 1) as simply (sel); (sel == 0) could be written as (~sel).

3 Structural Verilog

The examples given above use *behavioral models*; that is, you write logic equations to describe the manner in which the module should function, and the Verilog tools synthesize the best possible implementation for you. You can also write Verilog code that specifically indicates which gates should be used to implement a given function, much as you would do when drawing a circuit schematic. Such code uses a *structural model*, the code for which looks more like assembly language than C code.

As an example, we have rewritten the half and full adder macros above using structural Verilog code:

```
module add_half (a, b, s, cout);

    input  a, b;
    output s, cout;
    wire s, cout;

    xor x1 (s, a, b);
    and a1 (cout, a, b);

endmodule    // end of half adder module

module add_full (a, b, cin, s, cout);

    input a, b, cin;
    output s, cout;
    wire s, cout;
    wire ab, ac, bc;   // Intermediate AND gate outputs

    xor x2 (s, a, b, cin);
    and a2 (ab, a, b);
    and a3 (ac, a, cin);
    and a4 (bc, b, cin);
    or o1 (cout, ab, bc, ac);

endmodule    // end of full adder module
```

All of the standard logic gates (AND, OR, XOR, NOT, NAND, NOR) are available to you. To instantiate a gate in structural Verilog, you use the following syntax:

```
<gate type> <instance name> (<output name> <input list>);
```

<gate type> is one of the standard gate names. Each gate must have its own unique instance name, which is specified next, within a given module. The output signal name must be first in the parameter list; giving a multi-bit signal as the output will implicitly

instantiate multiple copies of the same type of gate, and the input bit widths must match the output bit width. The input list should generally be limited to reasonable numbers of inputs (for example, instantiating a 10-input AND gate may not be a good idea).

Note that all gate outputs must be declared as wires if they are not inputs or outputs. For example, let's say you're implementing the function $F = A + \overline{B}$ in structural Verilog, and you have A and B as inputs, and F as an output. You still need to declare a wire for the inverter output, so your code might look like this:

```
input A, B;
output F;
wire F;
wire B_not;

not n1 (B_not, B);
or o1 (F, A, B_not);
```

References

Much of this material was taken from the EECS 470 Verilog reference material, found at the following locations:

http://www.eecs.umich.edu/courses/eecs470/470VerilogDesign.pdf
http://www.eecs.umich.edu/courses/eecs470/synth.pdf
http://www.eecs.umich.edu/courses/eecs470/VRG470.pdf

Material was also taken from other Verilog tutorials found online, including:

The ASIC World Verilog tutorial by Deepak Kumar Tala, available at http://www.asic-world.com/verilog/veritut.html

"CSCI Computer Architecture Handbook on Verilog HDL," by Dr. Daniel C. Hyde, available at http://www.eg.bucknell.edu/~cs320/1995-fall/verilog-manual.html

"The Verilog Hardware Description Language," by Professor Don Thomas, available at http://www.ece.cmu.edu/~thomas/VSLIDES.pdf