Module Basics

module example(o1, i1, i2); // Declaration for a module named 'example' with 3 ports named 'o1', 'i1', and 'i2'
    input i1, i2; // Declaring ports 'i1' and 'i2' as inputs to the module
    output o1; // Declaring port 'o1' as an output of the module
    wire w1; // Declaring a single 1-bit wire called 'w1'
    wire w2, w3; // Declaring two 1-bit wires called 'w2' and 'w3'
    and a1(w1, i1, i2); // AND gate named 'a1', with 2 inputs 'i1' and 'i2' and with output 'w1'
    or  o1(w2, w1, w3); // In Verilog statement order does not matter, the value of 'w3' will be used as an output after 'o1' and 'a2'.
    and a2(o1, w2, w3); // example module named 'e1', filling in the inputs and outputs the same as they are declared above.
    nor o2(w3, w1, i1); // Using other modules works the same as declaring a gate.
endmodule

module example2(o1, i1);
    input i1, i2;
    output o1;
    wire w1;
    not n1(w1, i1);
    example e1(o1, i2, w1); // example module named 'e1', filling in the inputs and outputs the same as they are declared above.
endmodule
wire <id>, <id>, ...;
input <id>, <id>, ...;
output <id>, <id>, ...

and <id>(out, in1, in2, ..., inN);
or <id>(out, in1, in2, ..., inN);
xor <id>(out, in1, in2, ..., inN);
nand <id>(out, in1, in2, ..., inN);
nor <id>(out, in1, in2, ..., inN);
xnor <id>(out, in1, in2, ...., inN);
not <id>(out, in);

**Identifiers (names)**
- Must begin with alphabetic or underscore characters a-z A-Z _
- May contain the characters a-z A-Z 0-9 _ and $

**Logic Values (outputs)**
A wire has 4 possible outputs:
- '0' (low)
- '1' (high)
- 'z' (high impedance, typically occurs when no value was ever assigned to the wire)
- 'x' (unknown or uninitialized, sometimes occurs when you attempt to assign both a '1' and '0' to a wire at the same time)

**Testing**
module sc_test;

    reg a, b;                   // these are inputs to "circuit under test"
    // use "reg" not "wire" so can assign a value
    wire s, c;                  // wires for the outputs of "circuit under test"

    sc_block scl (s, c, a, b);  // the circuit under test

    initial begin               // initial = run at beginning of simulation
        // begin/end = associate block with initial
        $dumpfile("sc.vcd");    // name of dump file to create
        $dumpvars(0,sc_test);   // record all signals of module "sc_test"
    and sub-modules

    $finish;                // end the simulation
    end

    initial
        $monitor("At time %2t, a = %d b = %d s = %d c = %d", 
        $time, a, b, s, c);  // Anytime one of the parameter values changes, the
        // string will be printed to terminal. Uses C-like // `printf` syntax. (\%d` to format
    endmodule

Lab 2

More wires
wire [31:0] w1, w2; // Declares 2 32-bit buses of wires 'w1' and 'w1'
or o12(out, w1[31:16], w2[15:0]); // Selects the 16 most significant bits of the 'w1' bus and the // 16 least significant bits of the 'w2' bus to use in the or gate
assign x = w1[31]; // Connects the 32nd bit of the 'w1' bus to wire 'x'
assign y[3:0] = w1[12:9]; // Connects bits 12 through 9 (inclusive) of the 'w1' bus to the // bits 3 through 0 (inclusive) of the 'y' bus

Lab 3

Literal Integers
Literal integers are defined as size base value.

- size is the number of bits in the number.
- base represents the radix, legal values include 'b' for binary, 'o' for octal, 'd' for decimal, and 'h' for hexadecimal.
- value is the desired value in the given base

Examples:

8'hd7 // 8-bit number with value 0xd7 in hexadecimal (215)
5'b11001 // 5-bit number with value 11001 in binary (25)
10'd978 // 10-bit number with value 978 in decimal

Constants

`define ONE_HUNDRED 32'd100 // Define a constant named 'ONE_HUNDRED' that is a 32 bit number with decimal value 100
wire w1;
assign w1 = `ONE_HUNDRED; // Assign the value of 'ONE_HUNDRED' to 'w1'. Notice the back tick (`) before 'ONE_HUNDRED'

More Testing

`always #<number>` allows you to have something happen every 'number' of time steps:

always #1 A = !A; // Every time step, assign 'A' to '!A' (i.e. flip the bit)
always #2 B = !B; // Every 2 time steps, assign 'B' to '!B'
Lab 4

Boolean Expressions and Comparisons

Operators are similar to C. You may use the following bit-wise operators:

- AND = \&
- OR = |
- XOR = ^
- NAND = ~&
- NOR = ~|
- XNOR = ~^
- (unary) NOT = ~

Additionally, you may compare to wires using \(\text{==}\) which returns a 1-bit value, 0 if the wires' values differ, 1 if they are the same. Any width wire may be used, but all wires within an operation must be the same width. The only exception is \(\text{==}\) which produces a 1-bit value.

Examples:

```verilog
input [31:0] w1, w2, w3, w4, w5, w6, w7;
input w8;
assign w3 = w1 & w2;       // Assign value of 'w3' to bit-wise AND of 'w1' and 'w2'
assign w5 = ~w4;           // Assign value of 'w5' to bit-wise NOT of 'w4'
assign w8 = w6 == w7;      // If 'w6' and 'w7' have the same value, set 'w8' to 1, otherwise set to 0
```

Lab 5

Parametrized Verilog Modules

Verilog has a feature allowing modules to be parametrized, meaning that properties of the module can be defined or overridden by the user of said modules. For example, the muxes in our mux library have a width parameter specifying how wide the inputs/outputs of the mux are. This allows us to write one module that works for any desired input width, rather than having to rewrite it for all the possible widths we would use. The default width for these modules is 32, which is what we want most of the time, but occasionally we want a mux of different width, such as for parts of the decoder. Note that the input width has nothing to do with the number of inputs. In the examples below, we create a 32 bit wide mux and a 5 bit wide mux, but both of them have two inputs to choose from.

Examples:

```verilog
output [31:0] out;
output [4:0] out1;
wire [31:0] A, B;
wire [4:0] C, D;
wire control, control1;
// creates 32 bit wide (default width) mux
mux2v m1(out, A, B, control);
// creates 5 bit wide mux, overriding the default width parameter
mux2v #(5) m2(out1, C, D, control1);
```