CS61c Verilog Tutorial

J. Wawrzynek

April 14, 2002: Version 0.2

1 Introduction

There are several key reasons why we introduce and use hardware description languages (HDLs) in cs61c:

- They give us a text-based way to talk about designs with one another,
- They give us a way to simulate the operation of a circuit before we build it in silicon. It is usually easier to debug a model of the circuit rather than the real circuit in silicon,
- With special tools we can automatically translate our Verilog models to information needed for circuit implementation in silicon. This translation can take the form of simple partitioning, placement, and routing tools, and could also include logic synthesis—automatic generation of lower-level logic circuit designs from high-level specifications.

Two standard HDLs are in wide use, VHDL and Verilog. We use Verilog because it is easier to learn and use for most people because it looks like the C language in syntax. Also, it is widely used in industry. Furthermore, because the semantics of both are very similar, making a switch to VHDL from Verilog later not a problem.

Verilog is a general-purpose programming language that also includes special features for circuit modeling and simulation. In this course, we will employ only a simple subset of Verilog. In fact, we will focus just on those language constructs used for "structural composition"—sometimes also referred to as "gate-level modeling". These constructs allow us to instantiate primitive logic elements (logic gates) or subcircuits and connect them together with wires. With these contructs we can compose a model of any circuit that we wish, as long the primitive elements are ones included with Verilog. Structural composition is very similar to the process that we would go through, if we were to wire together physical logic gates in a hardware lab.

Even though we are primarily interested in *structural* Verilog, we will introduce some higher-level language constructs to help in testing our circuit models. The higher-level language constructs, called "behavioral constructs", are the ones that make Verilog a general purpose programming language (similar to C or Java). As we will see, the behavioral constructs are very convenient for automatically generating input to and checking output from our circuit models. In fact, an advantage of Verilog over other systems for modeling circuits, schematic capture for instance, is that it is powerful enough to also express complex testing procedures without resorting to a different language.

2 First Example

A multiplexor is a circuit used to select between a set of values. The multiplexor output takes on the value of in0 when select=0; otherwise it takes on the value of in1. We can express the 2-input multiplexor operation with the following boolean expression:

$$out = select \cdot in1 + \overline{select} \cdot in0$$

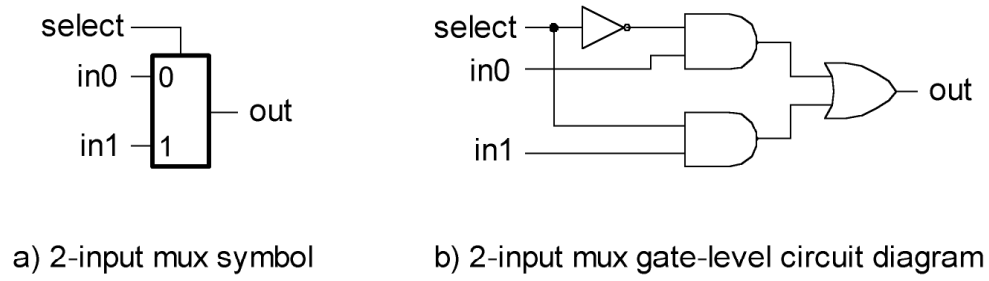

Figure 1: 2-input Multiplexor

This operation can be implemented with two 2-input AND gates, a 2-input OR gate, and an inverter, as shown in figure 1.

This circuit can be modeled in Verilog as follows:

```
module mux2 (in0, in1, select, out);
   input in0,in1,select;
   output out;
   wire s0,w0,w1;

   not
       (s0, select);
   and
       (w0, s0, in0),
       (w1, select, in1);
   or
       (out, w0, w1);

endmodule // mux2
```

The first thing to notice about this example is that the syntax is similar to that of C++ and Java, and other C-like languages. In fact, most of the same rules about naming variables (in this case inputs, outputs, and wires) all follow the same rules as in C. Unlike C, however, is that the body of mux2 is not made up of assignment statements. In this case, the body describes the connection of primitive logic elements (gates). It is important to understand that there is no real action associated with this description. In fact, it is more like defining a data-structure (struct in C) than it is like a program. The function that this circuit model assumes is a result of the function of the primitive elements and their interconnection.

Modules in Verilog are the basic mechanism for building hierarchies of circuits. Modules are defined and then instantiated in other module definitions. As with C functions, module definitions cannot be nested. A module definition begins and ends with the `module` and `endmodule` keywords, repectively. The pair of slashes ("//") signifies the beginning of a comment that extends to the end of the line. In this case, the "`// mux2`" comment after the `endmodule` was added automatically by emacs in "Verilog mode".

Following the keyword "`module`" is the user-defined name for that module, followed by a list of signals. These signals define the interface of the module to other modules; think of them as "ports".

When the module is instantiated, these port names are bound to other signals for interconnection with other modules. Each port can be defined as "`input`", "`output`", or some other types that we will not need in 61C. In mux2, after declaring the inputs and outputs, we define three additional signals, `s0`, `w0`, and `w1`. These additional signals will be used for connecting together the basic logic gates that will make up the model of mux2. The type of these signals is "`wire`"—the standard type used to make simple connections between elements.

Next is the body of the definition of mux2. It comprises a list of elements. This section could name other modules that we have defined, but in this case only instantiates primitive logic gates. These gates used here (NOT, AND, and OR) along with NAND, NOR, XOR, XNOR, BUF are predefined by Verilog. BUF is a single-input single-output gate (similar to NOT) that copies its input value to its output without inversion. The convention for built-in gates is that their output signal is the first port and the remaining ports inputs.

By following the signal names, the interconnection between the logic gates and there connection to the module ports should be apparent. An interesting exercise that you might try is to draw a schematic diagram for this circuit based on the Verilog and compare it to figure 1.

3 Testing our First Example

Given this definition of mux2, it is ready to be instantiated in other modules. However, before we do that, it is probably a good idea to test it. Several things could have gone wrong: we could have made a mistake typing in our definition, we could have made a mistake in our understanding of the correct circuit for implementing the multiplexor function, or perhaps we made an invalid assumption about how Verilog works.

Testing is a critical part of circuit design. Untested circuits are useless—they're a dime a dozen. The fun part of circuit design is coming up with the circuit configurations to achieve a desired function. The hard part is the testing. For most circuits, coming up with a good test program can take as much or more time than coming up with the orginal circuit to be tested. However, testing doesn't need to be difficult, and in Verilog there are several clever ways to make testing easier—and even fun!

In Verilog it is common practice to define a special module used specifically for testing another module. The special module is usually defined at the top level, having no ports. It is commonly referred to as a "test-bench". The name test-bench is an analogy to the laboratory work bench that houses the test equipment that we use in testing physical circuits.

A simple test bench for mux2 is shown below:

```
module testmux;
   reg a, b, s;
   wire f;
   reg expected;

   mux2 myMux (.select(s), .in0(a), .in1(b), .out(f));

   initial
     begin
            s=0; a=0; b=1; expected=0;
        #10      a=1; b=0; expected=1;
```

```
        #10 s=1; a=0; b=1; expected=1;
        #10 $finish;
      end
    initial
      $monitor(
        "select=%b in0=%b in1=%b out=%b, expected out=%b time=%d",
         s, a, b, f, expected, $time);
endmodule // testmux
```

The test-bench uses some of the constructs in Verilog that make it more like a progamming language. In particular, notice that the body of `testmux` uses assignment statements. These assignments are a way to set a signal to a particular logic value at a particular time. In general, these assignments can model the effect that a circuit can have on a signal, however, the assignment itself has no circuit directly associated with it. In other words, the assignment statement really only takes on meaning in the context of simulation. However, testing is all about simulation, so we will use assignments in our test-bench.

Because assignments are special, the left-hand side must be signals defined as type "`reg`". (Don't confuse these with the flip-flop-like circuits that we will be discussing later.) In this test-bench, we define the signals `a`, `b`, and `s` as type `reg` because these will be explicitly assigned values. The signal `f` will be connected to the output of our module to be tested, mux2. In fact, if we now skip down to the line that begins, "`mux2 myMux ...`" we see that we have instantiated mux2, giving it the local name "`myMux`". The local name is used to distinguish from other instances of mux2 that we might make.

What follows the instantiation of mux2 is a list of connections between our local signals and the ports of mux2. The syntax used here lists the ports of mux2 in arbitrary order, each one preceded by a ".", and followed by the name of a local signal in parentheses. Verilog also allows making connections between local signals and module ports by simply listing the names of the local signals (as was done with the primitive logic gates in the definition of mux2). However, we recommend you use the "dot" form because it allows you to list the ports in any order, making it easier to make changes later, and provides a reminder of exactly which signal is being connected to what port.

Now that we have established the connection between the local signals `a`, `b`, and `s`, and the ports of mux2, anything that we do to change the values of these local signals will instantaneously be seen by myMux.

In addition to the local signals defined for connecting to mux2, we have defined one additional signal, called "`expected`". This signal is used only to help provide useful output on the console, as will be seen below.

After instantiating mux2 and connecting it to local signals, we move on to defining the actions that will drive its inputs. Here we use the "`initial`" keyword to say that everything in the following block should be done once at the start of simulation. Verilog also includes a keywork "`always`" for defining actions that should be done every time a particular event occurs. In our case, what we do is to apply sets of input values (represented as `s`, `a`, and `b`) to mux2 and at the same time assign to "`expected`" what we expect to see at its output.

What probably looks particularly strange to you are the "`#n`"s preceding each assignment. These are used to move along the simulation clock. Unlike programming languages where each statement is executed after the previous without any regard for absolute time (only sequence is important) everything in Verilog happens at a particular time (well, simulated time really). By default, the unit of time is nanoseconds (ns). In testmux, the first set of assignments are made at 0 ns from the start of simulation,

the second at 10ns from the previous, etc. Without the "`#n`"s all these statements would occur at the same time (the start of simulation), causing undefined results. The important thing to remember about Verilog is that time does not move along until we do something to advance it. And without advancing time, no useful simulation can happen. In this example, new inputs are applied to our circuit every 10ns. The simulation is ended with the "`#10 $finish;`" line.

The final thing to specify in `testmux` is a way for us to observe what's happening. For this, we use another initial block that at the beginning of simulation starts a special built-in function for printing information to the console. "`$monitor`" watches the signals that are connected to it and whenever any of them change, it prints out a string with the all the signal values. The syntax of the string specification is similar to "`printf`" in C. Here the special format characters "`%b`" mean that the associated signal should be printed as a binary number. "`%d`" is used to print the value of the simulation time as a decimal number. The simulation time is available by using the special name "`$time`".

The result of executing testmux is shown below:

```
select=0 in0=0 in1=1 out=0, expected out=0 time=0
select=0 in0=1 in1=0 out=1, expected out=1 time=10
select=1 in0=0 in1=1 out=1, expected out=1 time=20
```