CS61c: Verilog Tutorial

J. Wawrzynek

November 17, 2003

1 Introduction

There are several key reasons why we use hardware description languages (HDLs):

- They give us a text-based way to describe and exchange designs,
- 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 the information needed for circuit implementation in silicon. This translation takes the form of partitioning and mapping to primitive circuit elements, element placement, and wire routing. The set of translation tools 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 language that 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 seem like 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.

It is important to keep in mind that, although Verilog has constructs that make it a look like a general purpose programming language, it is really only a hardware description language. Describing circuits

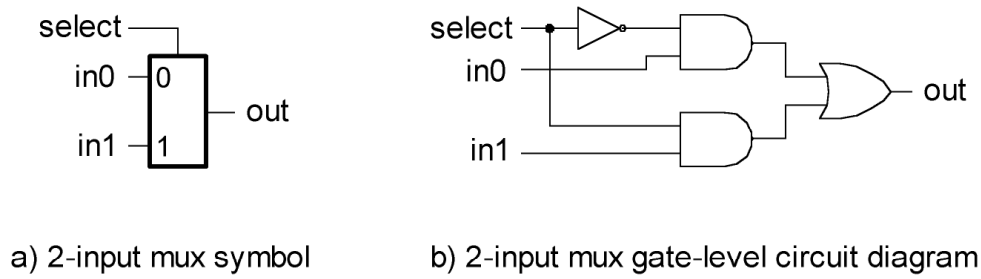

Figure 1: 2-input Multiplexor

in not equilavent to programming. The most effective way to use Verilog is to first figure out the circuit you want then figure out how to use Verilog to describe it.

2 Two-input Multiplexor 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:

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

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 the text editor 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. In addition to the gates used here, NOT, AND, and OR, Verilog predefines NAND, NOR, XOR, XNOR, and BUF. 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 are inputs. Except for NOT and BUF, these primitive gates can have any number of 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 the Multiplexor

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 configuration 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
        #0  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 state elements, registers and flip-flops.) 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 in the order that the ports where defined in the module header (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
```

3.1 Adding Delay to the Multiplexor Example

The multiplexor circuit that we described above contains no delay. You will notice from the simulation output listed above, that the output changes instantaneously with input changes. Primitive logic gates, and other circuit elements, in Verilog have any implicit logic delay. The delay much be specified as part of the circuit description. It is important to add delay to circuit elements to get a realistic simulation, and in sum cases to get correct operation.

Delay can be added to logic gates in Verilog by using the "`#`" operator and specifying a delay amount, in terms of simulation time units, such as "`not #1 (a, b);`".

Delay can be added to `mux2` as follows. Here we simply assume that the delay of each logic gate is 1ns.

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

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

endmodule // mux2
```

Now if we simulate this version of the multiplexor using the test bench that we previously defined, we would see the following output:

```
select=0 in0=0 in1=1 out=x, expected out=0 time=  0
select=0 in0=0 in1=1 out=0, expected out=0 time=  2
select=0 in0=1 in1=0 out=0, expected out=1 time= 10
select=0 in0=1 in1=0 out=1, expected out=1 time= 12
select=1 in0=0 in1=0 out=1, expected out=0 time= 20
select=1 in0=0 in1=0 out=0, expected out=0 time= 22
```

We now see twice the number of output lines printed. The lines with a time that is an even multiple of 10, correspond to changes in the input. The other lines (with time= 2, 12, and 22) correspond to changes in the output. You will notice that in the first line, at time=0, the output has value "x". The value "x" is used in Verilog to mean *undefined*. Because the way we assigned delays to the logic gates, it takes 2ns before the output takes on its new value.

From the above output, you might conclude that the mux delay is always 2ns. But in fact, worst case delay from the `select` input to the output is 3ns, because that path has three gates in series. We could verify the worst case delay by setting `select` to 1, `in0` to 1, `in1` to 0, and then changing `select` to 0. We would expect to the the output change from 0 to 1 after a 3ns delay.

4 Bit Vectors and Automatic Looping in Test-benches

Let's take a look at another test-bench for our 2-input mux:

```
//Test bench for 2-input multiplexor.
// Tests all input combinations.
module testmux2;
   reg [2:0] c;
   wire f;
   reg expected;
```

```
    mux2 myMux (.select(c[2]), .in0(c[0]), .in1(c[1]), .out(f));

    initial
      begin
        #0 c = 3'b000; expected=1'b0;
        repeat(7)
        begin
         #10
            c = c + 3'b001;
            if (c[2]) expected=c[1]; else expected=c[0];
        end
        #10 $finish;
      end
     initial
       begin
         $display("Test of mux2.");
         $monitor("[select in1 in0]=%b out=%b expected=%b time=%d",
                  c, f, expected, $time);
       end
endmodule // testmux2
```

This test-bench is designed to be a comprehensive test of the 2-input mux. Whereas the first test-bench we wrote tested only three input combinations, this new one preforms an exhaustive test, trying all possible input combinations. The 2-input mux has three inputs; so a complete set of tests needs to try eight different combinations. We could have simply extended the first test-bench to try all eight cases; however, that approach would be a bit tedious and even more so for circuits with more inputs. Remember the number of unique input combinations for a circuit with n inputs is 2^n.

The approach we use here is to generate all input combinations through looping and counting. We consider the three inputs to mux2, `a`, `b`, and `s` as three distinct bits of a 3-bit number, called `c`. The procedure starts by initializing `c` to all zeroes (`3'b000`) then sucessively increments `c` through all its possible values.

Now lets take a look at `testmux2` in more detail. Again we declare signals of type `reg` to be used on the left-hand side of assignment statements. The signal `expected` will again be used to store the expected output from the mux. The signal `c` is declared as a 3-bit wide signal. The special syntax "`[2:0]`" is used in a way similar to array declarations in high-level programming languages. A signal with width can be thought of as an array of bits. In Verilog, however, unlike C++ the declaration can also specify a naming convention for the bits. In this case the *range specifier*, "`2:0`" says that the rightmost bit will be accessed with "`c[0]`", the middle bit with "`c[1]`", and the leftmost with "`c[2]`".

After the signal declarations, mux2 is instantiated. Once again we establish the connections between the local signals of the test-bench and the module ports of mux2. Here we connect the bits of `c` to the three inputs of mux2, and the output of the mux to `f`.

The first initial block is the one that increments `c`. It begins by setting `c` to all zeroes, and `expected` to logic 0 (the expected output of a 2-input mux with zeroes at the inputs). The `repeat` contruct is used to loop for 7 interations and on each, advancing time by 10ns and incrementing `c` by 1 bit value.

Also included in the repeat block is the generation of the expected output value. Because the input values to mux2 are *automatically* generated in a loop, we need to *automatically* generate the value for

`expected`. By definition, we know that we can express the action of our multiplexor as "if select=0 then output=in0 else output=in1". The Verilog "`If`" contruct is used to express this relationship and assign the proper value to `expected`. After the initialization of `c` and seven iterations of the loop, the simulation is ended 10ns after the final loop iteration with the "`#10 $finish;`" line.

The second initial block is used to monitor the test results. Remember, all `initial` blocks start together at the beginning of simulation. In this case, we start off with the system command "`$display`". This command is similar to the `$monitor`, except that it prints out a string on the console when the command is executed, rather than every time the value of one of its input signals changes. In general `$display` can accept a output specifier string as can `$monitor`, but in this case we have passed it a fixed string.

The result of executing testmux2 is shown below:

```
Test of mux2.
[select in1 in0]=000 out=x expected=0 time=  0
[select in1 in0]=000 out=0 expected=0 time=  2
[select in1 in0]=001 out=0 expected=1 time= 10
[select in1 in0]=001 out=1 expected=1 time= 12
[select in1 in0]=010 out=1 expected=0 time= 20
[select in1 in0]=010 out=0 expected=0 time= 22
[select in1 in0]=011 out=0 expected=1 time= 20
[select in1 in0]=011 out=1 expected=1 time= 32
[select in1 in0]=100 out=1 expected=0 time= 40
[select in1 in0]=100 out=0 expected=0 time= 42
[select in1 in0]=101 out=0 expected=0 time= 50
[select in1 in0]=110 out=0 expected=1 time= 60
[select in1 in0]=110 out=1 expected=1 time= 62
[select in1 in0]=111 out=1 expected=1 time= 70
```

4.1 Using the "$strobe" System Command

The system command "`$monitor`" prints signal values whenever any of the signals it is connected to changes value. That can often result in a confusing looking display making it difficult to verify correct operation. As an alternative, system command "`$strobe`" can be used to print signal values at a particular time in the simulation. `$strobe` is defined to be the last action taken on any simulation step, so it is gauranteed to see the latest values of any signal it displays. In a typical simulation, inputs would be applied at one point in time, then after some delay sufficient to allow all the outputs to settle, `$strobe` would be to called to print out the input values and the associated outputs.

In the example below we have modified the mux test-bench to use `$strobe` instead of `$monitor`. Starting 5ns from the beginning of the simulation, `$strobe` is invoked every 10ns. The `$strobe` system command lives in its own `initial` block, which runs in parallel with the `initial` block used for setting input values. We could have put it all together in one block, but they were kept separate for clarity. The second `initial` block starts at time 0—the "`#5`" before the `$strobe` command is used to delay `$strobe` 5ns—then the "`#5 ;`" delays the start of the new `repeat` iteration for another 5ns, resulting in an iteration period of 10ns. The net effect is that each time the inputs are changed, 5ns later `$strobe` prints out the values of the inputs and the new output value. We chose the 5ns staggerng

between new inputs and printing the output, because we expect that the new outputs will have settled in less than 5ns (we know this to be true, because the worst case delay through the mux is 3ns).

```
//Test bench for 2-input multiplexor.
// Tests all input combinations.
module testmux2;
   reg [2:0] c;
   wire f;
   reg expected;

   mux2 myMux (.select(c[2]), .in0(c[0]), .in1(c[1]), .out(f));

   initial
     begin
       #0 c = 3'b000; expected=1'b0;
       repeat(7)
       begin
        #10
           c = c + 3'b001;
           if (c[2]) expected=c[1]; else expected=c[0];
       end
       #10 $finish;
     end
    initial
      begin
        $display("Test of mux2.");
        repeat(7)
          begin
            #5
            $strobe("[select in1 in0]=%b out=%b expected=%b time=%d",
               c, f, expected, $time);
            #5 ;
        end
      end
endmodule // testmux2
```

The result of executing testmux2 using `$strobe` instead of `$monitor` is shown below. Here it is clear to see that the circuit is functioning correctly, because on every line the output matches the expected output.

```
Test of mux2.
[select in1 in0]=000 out=0 expected=0 time=  5
[select in1 in0]=001 out=1 expected=1 time= 15
[select in1 in0]=010 out=0 expected=0 time= 25
[select in1 in0]=011 out=1 expected=1 time= 35
[select in1 in0]=100 out=0 expected=0 time= 45
[select in1 in0]=101 out=0 expected=0 time= 55
[select in1 in0]=110 out=1 expected=1 time= 65
```

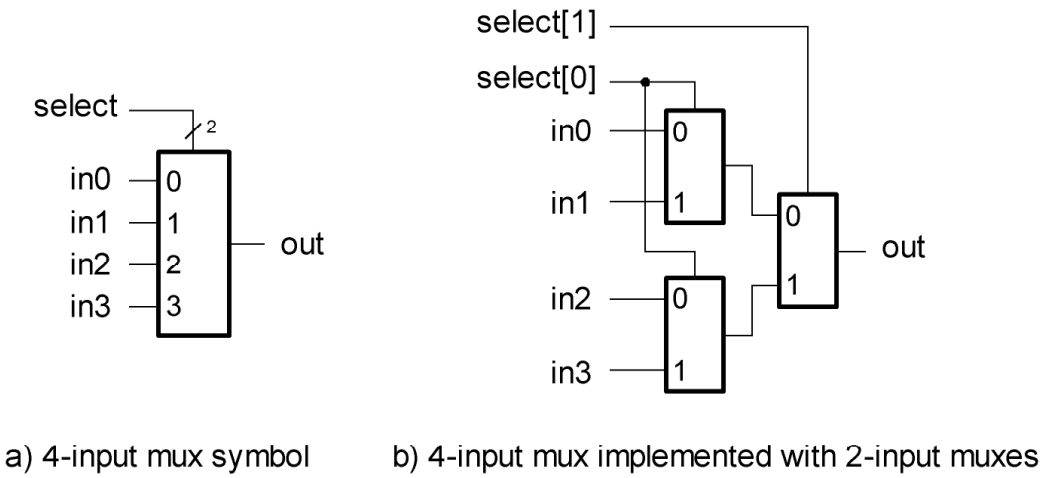

Figure 2: 4-input Multiplexor

5 Building a Circuit Hierarchy

An important tool in circuit design and specification is hierarchy. As you may have noticed, Verilog supports hierarchy in circuit specifications though the use of "module" definitions and instantiations. We have already seen the use of hierarchy in our discussion of test-benches. Each test bench can be considered a circuit that makes an instance of a sub-circuit—in this case, the circuit under test. To further investigate the use of hierarchy, let's take a look at the specification of a 4-to-1 multiplexor.

Any size multiplexor could be built up out of primitive AND and OR gates, as we have done for the 2-input multiplexor. However, a more convenient way to implement a bigger mux is from smaller ones. In the case of the 4-input mux, we will build it up out of 2-input muxes as shown in figure 2.

Given that we have already defined mux2, the Verilog description of mux4 is very simple:

```
//4-input multiplexor built from 3 2-input multiplexors
module mux4 (in0, in1, in2, in3, select, out);
   input in0,in1,in2,in3;
   input [1:0] select;
   output out;
   wire   w0,w1;

   mux2
     m0 (.select(select[0]), .in0(in0), .in1(in1), .out(w0)),
     m1 (.select(select[0]), .in0(in2), .in1(in3), .out(w1)),
     m3 (.select(select[1]), .in0(w0), .in1(w1), .out(out));
endmodule // mux4
```

The port list for mux4 includes the four data inputs, the control input, `select`, and the output, `out`. In this case, `select` is declared as a 2-bit wide input port—"`input [1:0] select;`". Two local signals, `w0` and `w1`, are declared for use in wiring together the subcircuits. Three instances of mux2 are

created, interconnected, and wired to mux4 input and output ports. Because that are no primitive gates in `mux4` we have no need to add explicit delay. The delay from the input of mux4 to its output will be a consequence of the delay through the instances of mux2. (In real circuits, the wire that interconnects subcircuits also introduces delay, however that level of detail is beyond our concern in cs61c.) At this point, mux4 is nearly ready to use in other modules—but not until we test it!

Once again we will test our new module exhaustively. In principle we could simplify the testing procedure by taking advantage of the fact the the subcircuit mux2 has already been tested, and only write tests to check the connections between the subcircuits. However, an exhaustive testing procedure is simple to write and verify and there are a reasonably small number of input combinations, even for a 4-input mux. A test-bench for mux4 is shown below:

```
//Test bench for 4-input multiplexor.
// Tests all possible input combinations.
module testmux4;
   reg [5:0] count = 6'b000000;
   reg a, b, c, d;
   reg [1:0] s;
   reg expected;
   wire f;

   mux4 myMux (.select(s), .in0(a), .in1(b), .in2(c), .in3(d), .out(f));

   initial
      begin
         repeat(64)
            begin
              a = count[0];
              b = count[1];
              c = count[2];
              d = count[3];
              s = count[5:4];
              case (s)
                2'b00:
                  expected = a;
                2'b01:
                  expected = b;
                2'b10:
                  expected = c;
                2'b11:
                  expected = d;
              endcase // case(s)
              #8
              $strobe(
              "select=%b in0=%b in1=%b in2=%b in3=%b out=%b, expected=%b time=%d",
                 s, a, b, c, d, f, expected, $time);
              #2 count = count + 1'b1;
          end
        $finish;
        end
endmodule
```

The testing procedure followed for testmux4 is very similar to that of testmux2. Here the signal `count` is used in place of `c` from testmux2. Four additional signals, `a`, `b`, `c`, and `d`, are declared and used simply to help in the coding. One significant difference between testmux2 and this new one is the construct used for setting `expected`. Here the Verilog `case` construct is used. The `case` construct is very similar to `switch` in C++. Also, as in C++, the function of a case can be achieved with a set of if-than-else statements, but the case is simpler and clearer.

Output is generated using the `$strobe` system command. Here it is put in the `repeat` loop and is delayed 8ns from the setting of the inputs, to allow sufficient time for the output to settle.

6 Modeling Clocks and Sequential Circuits

Thus far in this tutorial we have considered only combinational logic circuits. Now we turn our attention to sequential logic circuits.

Sequential circuits are circuits and include state elements—usually flip-flops or registers. Flip-flops and thus registers are built from transistors and logic gates, as are combinational logic circuits, and therefore we could model them as we did combinational logic. However, our focus in CS61c is not on the internal details of registers; we are really only interested in their function or behavior. To keep our Verilog specifications easier and to speed up the simulations, we will abstract the details of flip-flops and registers and model them using high-level behavioral constructs in Verilog.

Below is a behavioral model of a 32-bit wide register. This one is a *positive edge triggered* design; it captures the value of the input `D` on the rising edge of the clock and sends it to the output `Q` after the clk-to-q delay, in this case 1ns. The input port named `RST` supplies a reset signal. If `RST` is asserted on the positive edge of the clock, the register is reset to all 0's. The internal operation of this module is not important for our purposes, but with a few minutes of study you will probably be able to understand how it works. The most important consideration for our purposes is its list of input/output ports. Along with `RST`, the other ports are `CLK`, `Q`, and `D`. `D` is the data input, `Q` the data output, and `CLK` the system clock.

```
//Behavioral model of 32-bit Register:
// positive edge-triggered,
// synchrounous active-high reset.
module reg4 (CLK,Q,D,RST);
   input [3:0] D;
   input  CLK, RST;
   output [3:0] Q;
   reg [3:0] Q;
   always @ (posedge CLK)
     if (RST) #1 Q = 0; else #1 Q = D;
endmodule // reg4
```

The system clock is a globally supplied signal in all synchronous logic systems. In physical hardware the signal is generated from a special clock oscillator based on a *crystal*—a very stable oscillation

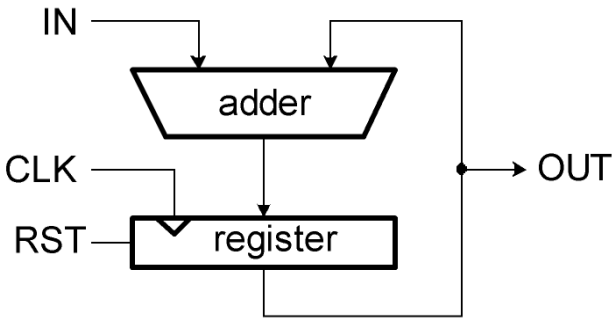

Figure 3: Accumulator Circuit

source. Verilog does not supply a clock signal automatically; we must find a way to generate a oscillating signal within our specification. A standard way to do this is to assign a signal to an inverted version of itself, after the appropriate delay. For example, after declaring `CLK` as type reg:

```
initial
  begin
     CLK=1'b0;
     forever
       #5 CLK = ~CLK;
  end
```

`CLK` begins at logic 0 then changes to logic 1 after 5 ns then back to 0 after another 5ns, etc. This continues until the end of the simulation. The result is a signal with an oscillation period of 10ns. Here we assume that some other part of the Verilog specification is responsible for ending the simulation, so we can allow the clock to oscillate "forever".

Now that we have a clock signal and a register to connect it to, we are ready to specify a sequential logic circuit. Sequential circuits are really nothing other than interconnected instances of combinational logic blocks and state elements. Everything that we have discussed thus far concerning making instances of modules and wiring them together applies as well to sequential logic.

Let's take a look at a circuit useful for adding lists of numbers, called an accumulator. The block diagam for this circuit is shown in figure 3. The reset signal, RST, is used to force the register to all 0's, then on each cycle of the clock the value on IN is added to the value in the register and the result stored back into the register.

The Verilog description for the accumulator circuit is shown below:

```
//Accumulator
module acc (CLK,RST,IN,OUT);
   input CLK,RST;
   input [3:0] IN;
   output [3:0] OUT;

   wire [3:0]  W0;

   add4 myAdd (.S(W0), .A(IN), .B(OUT));
   reg4 myReg (.CLK(CLK), .Q(OUT), .D(W0), .RST(RST));
endmodule // acc
```

This module definition assumes that we will also include a definition of a module called `add4` with the following port list:

```
module add4 (S,A,B);
```

This module is a combinational logic block that forms the sum of the two 4-bit binary numbers `A` and `B`, leaving the result in `S`. As part of its module specification, the input/output delay for the add4 block is defined as 4ns.

A test-bench for the accumulator circuit is shown below:

```
module accTest;
   reg [3:0] IN;
   reg       CLK, RST;
   wire [3:0] OUT;

   acc myAcc (.CLK(CLK), .RST(RST), .IN(IN), .OUT(OUT));

   initial
     begin
        CLK=1'b0;
        repeat (20)
          #5 CLK = ~CLK;
     end
   initial
     begin
        #0 RST=1'b1; IN=4'b0001;
        #10 RST=1'b0;
     end
   initial
     $monitor("time=%0d: OUT=%1h", $time, OUT);
endmodule // accTest
```

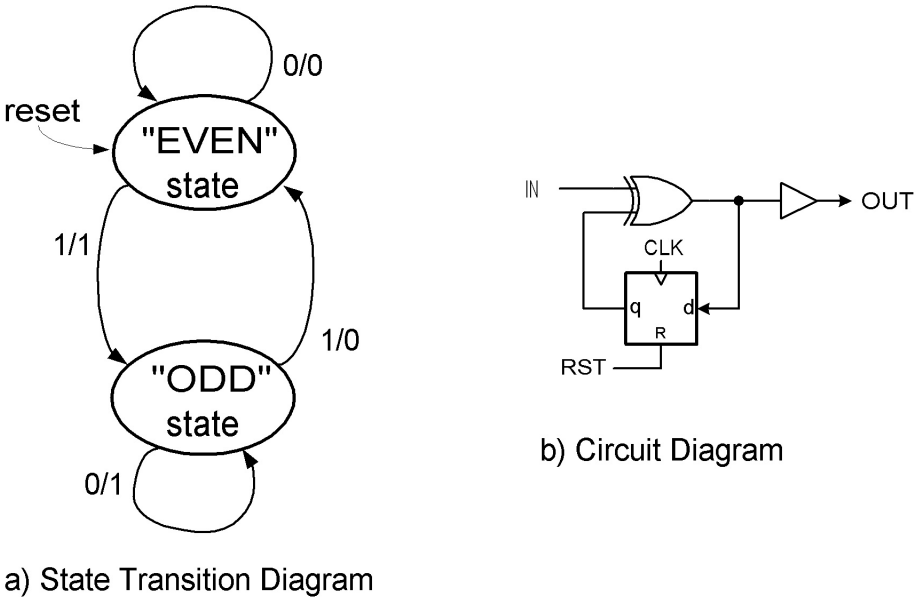

Figure 4: Bit-serial Parity Checker

This one works by first asserting the reset signal for one clock cycle, in the second `initial` block. At the same time the input is set to the value of decimal 1 (0001 in binary) and held at that value for the remainder of the simulation. Meanwhile in the first `initial` block the clock signal is forced to oscillate for 10 cycles. The output should be a sequence of numbers 0,1,2,... for 10 clock cycles.

7 Specifying Finite State Machines

A common type of sequential circuit used in digital design is the finite state machine (FSM). A simple FSM specification in Verilog is shown below. This example is a bit-serial parity checker; it computes the parity of a string of bits sequentially, finding the exclusive-or of all the bits (Parity of "1" means that the string has an *odd* number of 1's.) On every clock cycle, the circuit accepts a new input bit and outputs the parity of all the bits seen since the previous reset. The reset signal forces the FSM back to the EVEN state. The state transition diagram and the hardware implementation for the bit-serial parity checker is shown in figure 4.

Our Verilog specification of `parityChecker` is preceded by the specification of a D-type flip-flop. This is essentially a one-bit wide version of the register used in our previous example.

The `parityChecker` module is the FSM specification. It instantiates one flip-flop used to hold the present state of the FSM. We have chosen to represent the "EVEN" state with logic 0, and the "ODD" state with logic 1. The module also instantiates a single exclusive-or gate, used to compute the next state and output values. (The Verilog name for the exclusive-or gate is "`xor`".) We preceed the specification of the xor gate with "`#1`" to indicate a 1ns logic delay for the xor. One extra "gate", `buf`, is needed to connect the output of the flip-flop to the output port of the module, `OUT`. The Verilog `buf` gate is similar to a `not` gate (inverter) in that it has a single input and a single output, except that it does not invert its

input signal, but merely passes it through to its output. (This gate serves no real function, other than to get around a limitation in Verilog which prohibits feeding outputs back internally within a module). We assume that it has no delay.

```
//Behavioral model of D-type flip-flop:
// positive edge-triggered,
// synchronous active-high reset.
module DFF (CLK,Q,D,RST);
   input D;
   input CLK, RST;
   output Q;
   reg Q;
   always @ (posedge CLK)
     if (RST) #1 Q = 0; else #1 Q = D;
endmodule // DFF

//Structural model of serial parity checker.
module parityChecker (OUT, IN, CLK, RST);
   output OUT;
   input  IN;
   input  CLK, RST;
   wire   currentState, nextState;

   DFF state (CLK, currentState, nextState, RST);
   #1 xor (nextState, IN, currentState);
   buf (OUT, nextState);

endmodule // parity
```

8 Testing Finite State Machines

A test-bench for the parity checker module is shown below. The best testing strategy for finite state machines is to test the response of the FSM to every input in every state. In other words, we would like to force the FSM to traverse every arc emanating from every state in the state transistion diagram. The comment block in the test-bench preceding the specification lists the set of tests needed to completely test the FSM. After reset, each test forces the circuit to traverse one arc in the state transistion diagram. The expected output is written in square brackets, “[ ]”.

In these tests we assume that the reset signal works correctly. This is a good assumption, given that the reset action is implemented as a built-in input to the flip-flop, and can assume that the behavior of the flip-flop was independently verified.

```
//Test-bench for parityChecker.
//
//             Reset to EVEN.                    RST=1   [OUT=x]
// Set IN=0.   Self-loop in EVEN.          IN=0, RST=0   [OUT=0]
// Set IN=1.   Arc to ODD.                 IN=1          [OUT=1]
// Set IN=0.   Self-loop in ODD.           IN=0          [OUT=1]
// Set IN=1.   Arc back to EVEN.           IN=1          [OUT=0]
//
module testParity0;
   reg IN;
   wire OUT;
   reg     CLK=0, RST;
   reg     expect;

   parityChecker myParity (OUT, IN, CLK, RST);

   always #5 CLK = ~CLK;

   initial
     begin
              RST=1;
    #10 IN=0; RST=0; expect=0;
    #10 IN=1; RST=0; expect=1;
    #10 IN=0; RST=0; expect=1;
    #10 IN=1; RST=0; expect=0;
    #10 $finish;
     end

    always
      begin
         #4 $strobe($time," IN=%b, RST=%b, expect=%b OUT=%b",
                  IN, RST, expect, OUT);
         #6 ;
      end
endmodule // testParity0
```

This module demonstrates a variation on clock generation. In this example, the clock signal `CLK` is initially set equal to 0 when it is declared. Therefore, when simulation begins `CLK` will be equal to 0. The block specified by "`always #5 CLK = CLK;`" continuously inverts the value of `CLK` every 5ns, resulting in a clock period of 10ns.

Inputs are forced to change every 10ns, synchronously with the high-to-low transition of the clock.

Once again, we use the `$strobe` system command to print the results of our simulation. The first invocation of `$strobe` occurs at 4ns, and is repeated every 10ns, just before each rising edge of the clock. Execution of the `testParity0` results in the following:

```
 4 IN=x, RST=1, expect=x OUT=x
14 IN=0, RST=0, expect=0 OUT=0
24 IN=1, RST=0, expect=1 OUT=1
34 IN=0, RST=0, expect=1 OUT=1
44 IN=1, RST=0, expect=0 OUT=0
```

The circuit functions correctly and passes all tests.