CS61c Verilog Tutorial

J. Wawrzynek

April 14, 2002: Version 0.2

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
       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 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 successively advance time by 10ns and on each time step increment `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=0 expected=0 time=0
[select in1 in0]=001 out=1 expected=1 time=10
[select in1 in0]=010 out=0 expected=0 time=20
[select in1 in0]=011 out=1 expected=1 time=30
[select in1 in0]=100 out=0 expected=0 time=40
[select in1 in0]=101 out=0 expected=0 time=50
[select in1 in0]=110 out=1 expected=1 time=60
[select in1 in0]=111 out=1 expected=1 time=70
```

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:

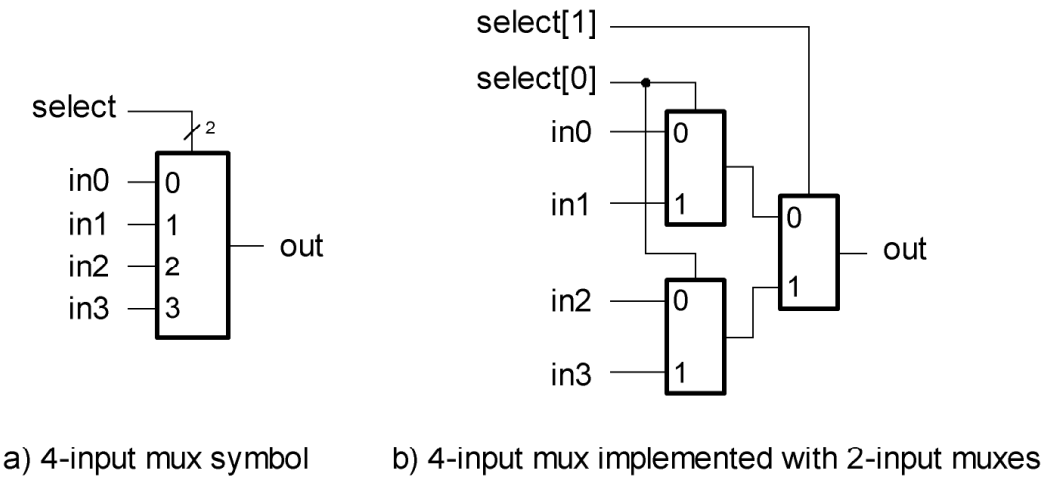

Figure 1: 4-input Multiplexor

```
//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. 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
$monitor("select=%b in0=%b in1=%b in2=%b in3=%b out=%b, expected=%b time=%d",
s, a, b, c, d, f, expected, $time);
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)
      #10 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.

6 Modelling 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.

A distinguishing characteristic of sequential circuits is that they 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. The input 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) Q = 0; else 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 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 itself, after the appropriate delay. For example, after declaring `CLK` as type reg:

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

`CLK` begins at logic 0 then changes to logic 1 after 1 ns then back to 0 after another 1ns, etc. This continues until the end of the simulation. The result is a signal with an oscillation period of 2ns. 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. Recall that sequential circuits are really nothing other than interconnected instances of

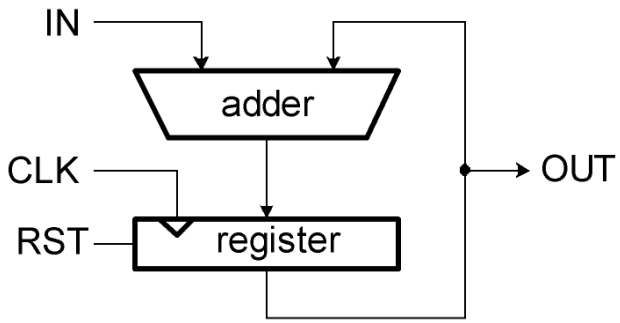

Figure 2: Accumulator Circuit

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 list 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`.

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
```

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.