CS61c Verilog Tutorial

J. Wawrzynek

April 24, 2002: Version 0.2

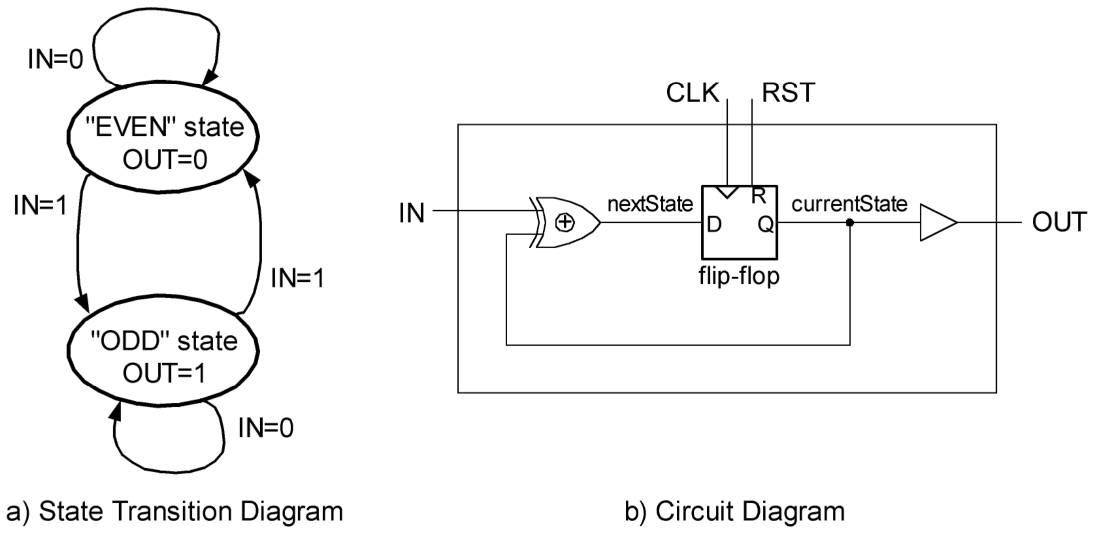

Figure 1: Bit-serial Parity Checker

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 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 is a simple module that instantiates one flip-flop (which holds the parity of the string seen since the previous reset) and a single exclusive-or gate. (The Verilog name for the exclusive-or gate is “`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.

```
//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) Q = 0; else 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);
   xor (nextState, IN, currentState);
   buf (OUT, currentState);

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. Each test forces the circuit to traverse one arc in the state transistion diagram.

```
//Test-bench for parityChecker.
// Set IN=0.   Assert RST. Verify OUT=0.   IN=0, RST=1   [OUT=0]
// Keep IN=0.  Verify no output change.    IN=0, RST=0   [OUT=0]
// Assert IN.  Verify output change.       IN=1          [OUT=1]
// Set IN=0.   Verify no output change.    IN=0          [OUT=1]
// Assert IN.  Verify output change.       IN=1          [OUT=0]
// Keep IN=1.  Back to ODD.                IN=1          [OUT=1]
// Assert RST. Verify output change.       IN=0, RST=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
        IN=0; RST=1; expect=0;
    #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 IN=1; RST=0; expect=1;
    #10 IN=0; RST=1; expect=0;
    #10 $finish;
     end

   initial
     $monitor($time," IN=%b, RST=%b, expect=%b OUT=%b", IN, RST, expect, OUT);
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 `$monitor` system command to print the results of our simulation. Execution of the `testParity0` results in the following:

```
    5 IN=0, RST=1, expect=0 OUT=0
    10 IN=0, RST=0, expect=0 OUT=0
    20 IN=1, RST=0, expect=1 OUT=0
    25 IN=1, RST=0, expect=1 OUT=1
    30 IN=0, RST=0, expect=1 OUT=1
    40 IN=1, RST=0, expect=0 OUT=1
    45 IN=1, RST=0, expect=0 OUT=0
    50 IN=1, RST=0, expect=1 OUT=0
    55 IN=1, RST=0, expect=1 OUT=1
    60 IN=0, RST=1, expect=0 OUT=1
    65 IN=0, RST=1, expect=0 OUT=0
```

The circuit does function correctly, although it may not be apparent. Several factors complicate the printed output. First, you will notice that a line of output is printed on every time step that is a multiple of 10. These lines correspond to changes we applied to the inputs, `RST` and `IN`, or to `expected`. However, output is also printed sometimes in between the even multiples of 10—for instance at 25 and 45. These lines correspond to changes in the circuit output, `OUT`. In this circuit `OUT` is simply the output of the state flip-flop. The flip-flop, being a positive edged triggered design, changes its output on the rising edge of the clock, in this case every 10ns starting at 5ns.

Here, once again is the printed output resulting from the execution of testparity0:

```
    5 IN=0, RST=1, expect=0 OUT=0     OUT X→0
    10 IN=0, RST=0, expect=0 OUT=0    RST 1→0
    20 IN=1, RST=0, expect=1 OUT=0    IN 0→1, expect 0→1
    25 IN=1, RST=0, expect=1 OUT=1    OUT 0→1
    30 IN=0, RST=0, expect=1 OUT=1    IN 1→0
    40 IN=1, RST=0, expect=0 OUT=1    IN 0→1
    45 IN=1, RST=0, expect=0 OUT=0    OUT 1→0
    50 IN=1, RST=0, expect=1 OUT=0    expect 0→1
```

```
55 IN=1, RST=0, expect=1 OUT=1    OUT 0→1
60 IN=0, RST=1, expect=0 OUT=1    IN 1→0, RST 0→0, expect 1→0
65 IN=0, RST=1, expect=0 OUT=0    OUT 1→0
```

In this version, we have annotated each line with the change that resulted in that line being printed.

The other confusing aspect of the printed output is the fact that the expected circuit output value, `expected`, and the actual output value, `OUT`, appear on different lines. The expected output value appears with it or the input values change, and the actual circuit output appears 5ns later, if it changes from its previous value, or 10ns later it it doesn't change.

Obviously it is desireable to display the circuit output on the same line as the expected value and to display it on every cycle whether or not it changes. To achieve this type of display, we need to introduce several new system commands.

9 Other Ways to Display Output

The system command "`$strobe`" can be used to print signal values at a particular time in the simulation. In the example below we invoke `$strobe` on every edge of the clock by executing it every 5ns, starting 5ns from the beginning of 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.

```
module testParity1;
   reg IN;
   wire OUT;
   reg     CLK=0, RST;
   reg     expect;

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

   always #5 CLK = ~CLK;

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

   initial
     #0 $strobe($time," IN=%b, RST=%b, expect=%b OUT=%b",
        IN, RST, expect, OUT);

   always
     #5 $strobe($time," IN=%b, RST=%b, expect=%b OUT=%b",
        IN, RST, expect, OUT);
endmodule // testParity1
```

The result of executing `testParity1` is shown below. In this case we have separated the lines into groups of two and annotate each with the the new information being displayed by each.

```
    0 IN=0, RST=1, expect=0 OUT=X    Inputs and “expect”.
    5 IN=0, RST=1, expect=0 OUT=0    New output.

    10 IN=0, RST=0, expect=0 OUT=0   Inputs and “expect”.
    15 IN=0, RST=0, expect=0 OUT=0   New output.

    20 IN=1, RST=0, expect=1 OUT=0   Inputs and “expect”.
    25 IN=1, RST=0, expect=1 OUT=1   New output.

    30 IN=0, RST=0, expect=1 OUT=1   Inputs and “expect”.
    35 IN=0, RST=0, expect=1 OUT=1   New output.

    40 IN=1, RST=0, expect=0 OUT=1   Inputs and “expect”.
    45 IN=1, RST=0, expect=0 OUT=0   New output.

    50 IN=1, RST=0, expect=1 OUT=0   Inputs and “expect”.
    55 IN=1, RST=0, expect=1 OUT=1   New output.

    60 IN=0, RST=1, expect=0 OUT=1   Inputs and “expect”.
    65 IN=0, RST=1, expect=0 OUT=0   New output.
```

This display is easier to understand because now we see two lines of output for every clock cycle—one that corresponds to setting inputs and one for checking circuit output.

Another change that we might want to make in how we display the circuit behavior is to show the circuit input values and the resulting output on the same line. In the example below we use the system command `$write` to print the values of `IN`, `RST`, and `expected`. `$write` is similar to the command `$display`, used in an earlier example, in that it prints the value of signals when it is executed as opposed to when the values change. However, `$write` does not print a newline. In this example, `$write` is executed every 10ns starting at 0, and `$strobe` is executed every 10ns starting at 5. Although they execute at different times their results appear on the same line.

```
module testParity2;
   reg IN;
   wire OUT;
   reg     CLK=0, RST;
   reg     expect;

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

   always #5 CLK = ~CLK;

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

   always
     begin
       $write($time," IN=%b, RST=%b, expect=%b ",
           IN, RST, expect);
    #5 $strobe($time," OUT=%b",
           OUT);
    #5 ;
     end
endmodule // testParity2
```

The results of executing `testParity2` are shown below:

```
0 IN=0, RST=1, expect=0 5 OUT=0
10 IN=0, RST=0, expect=0 15 OUT=0
20 IN=1, RST=0, expect=1 25 OUT=1
30 IN=0, RST=0, expect=1 35 OUT=1
40 IN=1, RST=0, expect=0 45 OUT=0
50 IN=1, RST=0, expect=1 55 OUT=1
60 IN=0, RST=1, expect=0 65 OUT=0
```