0. And *vice versa*. However, the more specific you get in capturing these conditions the closer you get to diagnosing a specific hardware implementation. After all, the addition inside the system might be done bit-serially or byte-serially. If you knew that you might come up with a different set of cover points. Remember though that verification is meant to work from a design's specification and not from its implementation. The actual internal organization of the adder might not be specified.

If a cover point does not explicitly define the number of bins or the value ranges that are counted in the bins, SystemVerilog will automatically define the bins and their value or value range. If the variable being counted is an enumerated variable, then the number of bins is the cardinality of the enumeration. If the variable is not an enumeration, then the number of bins for an M-bit variable is the minimum of 2^M or the value set by the auto_bin_max option (whose default is 64). This option would be specified as

```
option.auto_bin_max = 10;
```

and would apply to the whole cover group. It would be specified along with the other option on line 2 of Example 10.6.

```
1  covergroup sumItUp  @(posedge ck);
2    option.at_least = 15;
3    bitView: coverpoint sum
4      {
5      wildcard bins b0 =  { 16'b????_????_????_???1 };
6      wildcard bins b1 =  { 16'b????_????_????_??1? };
7      wildcard bins b2 =  { 16'b????_????_????_?1?? };
8      wildcard bins b3 =  { 16'b????_????_????_1??? };
9      wildcard bins b4 =  { 16'b????_????_???1_???? };
10     wildcard bins b5 =  { 16'b????_????_??1?_???? };
11     wildcard bins b6 =  { 16'b????_????_?1??_???? };
12     wildcard bins b7 =  { 16'b????_????_1???_???? };
13     wildcard bins b8 =  { 16'b????_???1_????_???? };
14     wildcard bins b9 =  { 16'b????_??1?_????_???? };
15     wildcard bins b10 = { 16'b????_?1??_????_???? };
16     wildcard bins b11 = { 16'b????_1???_????_???? };
17     wildcard bins b12 = { 16'b???1_????_????_???? };
18     wildcard bins b13 = { 16'b??1?_????_????_???? };
19     wildcard bins b14 = { 16'b?1??_????_????_???? };
20     wildcard bins b15 = { 16'b1???_????_????_???? };
21     }
22 endgroup: sumItUp
```

Example 10.6 — Coverpoint to count when specific bits are set to 1

10.2.3 Using Assertions to Trigger Cover Groups

To this point, we haven't considered whether the value of sum is correct. Although it is checked on line 42 of Example 10.4, the cover groups of Example 10.1, Example 10.5 and Example 10.6 will count the incorrect value. Generally, you only want to be counting based on correct values in the system.

These previous cover groups have only counted based on the value of sum at the clock's positive edge, or in the case of Example 10.5, when done was also asserted. Another option for triggering a cover group is to override its sampling function. This sampling function can then be called, for instance, from an assertion that is finishing or from a cover statement.

Consider the changes or additions to the above examples that are shown in Example 10.7. Lines 1-2 shows how to specify a function call that will cause the cover points in a cover group to count ("with" is a keyword, "sample" is not). This statement could be substituted for the first line of Example 10.6. Only when the sample function is called will the

```
1  covergroup Cover_sumItUp with function
2                sample(logic[15:0] sum);
3  ...
4  property checkSumWithTotal;
5    bit [15:0] myTotal;
6    @(posedge ck) (~go_l, myTotal = inA) |=>
7        (~done, myTotal += inA) [*1:10] ##1
8        (done && (sum == myTotal), sup.sample(sum));
9  endproperty
10
11 SumIncorrect: assert property (checkSumWithTotal)
12     else $error("Sum value incorrect");
```

Example 10.7 — Overriding the sample function

cover group be activated to count. This example shows only one value being passed when sample is called, however multiple values can be passed.

The sample function can be called from the assertion shown on lines 4-9. This assertion could be included in the previous example to check that sum is the correct value. The assertion waits for go_l to be asserted and then copies the input inA into the local variable myTotal. Then at each successive clock edge, as long as done is not asserted, it sums up the inputs into myTotal. Finally, when done is asserted and sum is equal to myTotal, the assertion passes. As a result of passing, the sample function is called, passing the final value of sum to the cover group as shown on line 8. Thus, only correct values are counted by the cover group.

Note that the assertion is working with values from the preponed region. In the previous example, the values being sampled were those loaded at the positive edge of the clock. Over the whole simulation run, the same values of sum are passed to the cover group, but this serves to illustrate that cover groups can be called from any of the timing regions of the simulation kernel.

An alternate approach to using assertions to initiate calls to overridden sample functions is to call them from a cover statement as shown in Example 10.8. The differences with respect to Example 10.7 is that the assert doesn't call the sample function (see line 5), and a cover statement has been added, lines 11-12. When the assertion passes correctly, the cover statement calls the overridden sample function with the value to be counted. Note that it must specify the preponed value by using the $sampled(sum). This is because the cover statement's pass statement is not using values from the preponed region (as the assertion statement is). In this example, the cover statement executes in the reactive region and thus the results of non-blocking assignments in the design have been resolved. Checking back to the sumItUp thread, we see that when the input inA is zero, done is asserted and sum is loaded with zero at then next clock edge. Thus the cover statement would pass sum (equal to 0) to the sample function if the code hadn't specified the $sampled value of sum.

```
1   property checkSumWithTotal;
2     bit [15:0] myTotal;
3     @(posedge ck) (~go_l, myTotal = inA) |=>
4           (~done, myTotal += inA) [*1:10] ##1
5           (done && (sum == myTotal));
6   endproperty
7
8   SumIncorrect: assert property (checkSumWithTotal)
9       else $error("Sum value incorrect");
10
11  Cover_Correct_sum_Values: cover property (checkSumWithTotal)
12      sup.sample($sampled(sum));
```

Example 10.8 — Using a cover Statement to Call the sample Function.

ly through its state transitions based on the values of in1 and in2. At some point, all of the transitions will be covered, although it may take awhile given that the system has to get to some "far away" transitions, such as a8. If it doesn't get there in 300 clock cycles, you will need to increase that number in the clock generator.

```
67    initial begin
68      InputTwiddle i = new;
69      r = 0;
70      #1 r = 1;
71      #1 r = 0;
72
73      while (fcover.get_coverage() < 100)
74        begin
75          assert (i.randomize());
76          {in1, in2} <= i.ins;
77          invals <= i.ins;
78          @(posedge ck);
79      end
80    end
81  endprogram
```

Example 10.11 — Testbench to Cover the Transition Bins

Note that this example doesn't check the correctness of the implementation, it just checks that each of the transitions was followed twice.

10.3.3 Using Cross Coverage Bins

Another option for binning information in a cover group is to create a *cross* cover as shown on line 4 of Example 10.12. A cross cover creates a cross product of bins using either two variables or, in this case, a cross product between variable invals and cover point st.

```
1     covergroup fsmCross @(posedge ck);
2       option.at_least = 2;
3       coverpoint st;
4       cross invals, st;
5     endgroup
6
7     fsmCross crossCover = new;
```

Example 10.12 — Cover Group with a cross

The purpose behind using a cross cover point is to count how many times the values of one of the two variables pairs with each of the possible values of the other variable. In this example one of the variables is the input vector, invals, and the other is the state variable, st. Thus, the cross is counting how many times we're in each of the states and, for each of the states, how many times we see each of the possible input combinations. Knowing that the system has been in each of the states with each of the possible input vectors on the inputs, we can be confident that the FSM's operation is being covered.

To use the cover group of Example 10.12 with the design of Example 10.9, the testbench program of Example 10.10 would be altered to contain cover group fsmCross instead of cover group fsm, and the while loop of Example 10.11 (line 73) would be conditioned by (crossCover.get_coverage < 100).

This examples requires more discussion though. In the previous examples, bins have been used to define how to count the values or transitions of the cover point variable. In this case, bins have not been defined; thus, SystemVerilog automatically defines them. Since st is an enumerated variable, a bin for each of the enumeration values is automatically defined. These bins are shown at the top of Figure 10.2.

The cross declaration can be defined between either two variables, or between a variable and a cover point. Here, on line 4, it is between a cover point and a variable. In this situation, SystemVerilog implicitly defines a cover point for the two-bit variable invals and

The *randc* modifier indicates that new values will be selected in a cyclic manner, cycling randomly through all values in the range before a repeated value is seen. This is randomization *without replacement*. Even though this is a logic variable, the random numbers will not include x and z bits.

In a procedural block, an object of the above class can be instantiated and the variables randomized as shown below.

```
5   bit     [4:0] outputA;
6   logic   [8:0] outputB;
7   bit           condition, clk;
8   testVectors tv = new;
9
10  initial begin
11    while (condition) begin
12      @(posedge clk);
13      assert (tv.randomize()) else $error ("OOPS, randomize didn't work");
14      outputA <= tv.fiveBits;
15      outputB <= tv.nineBits;
16    end
17  end
```

In this example several variables are declared and then a testVectors object tv is declared on line 8; new is called to instantiate it. Then, within the initial block, a while loop will keep repeating until condition is no longer asserted. Each iteration of the while loop is guarded by a clock edge. At that clock edge, the call to the randomize method on line 13 will cause the two object variables to take on new random values. On lines 14-15 these are assigned to other variables. Assuming that these variables are outputs of the testbench and inputs to the design, non-blocking assignments are used.

Thus, the tv.randomize() method call is a call to generated new values for the random variables in the tv object. This method returns a Boolean indicating whether it was successful or not. If not, the $error task is called. The next section will discuss why the return value of randomize needs to be checked. (An if statement can be used in place of the assert if you are not familiar with assertions.)

Randomization can be used with enumerated types. In the example below, a call to randomize object fc will result in myRainbow randomly taking on one of the values of the enumeration.

```
1   typedef enum
2       bit[2:0] {Red, Orange, Yellow, Green, Blue, Indigo, Violet} roy_g_biv_t;
3
4   class funnyColors;
5     rand  roy_g_biv_t  myRainbow;
6   endclass
7
8   funnyColors  fc = new;
```

7.5.2 Constraints on Random Numbers

Access to random variables is useful but being able to constrain their values provides a means to more closely control what is being tested. SystemVerilog provides several means of constraint specification.

One approach to constraining a random value is by putting a constraint on the randomize method:

```
tv.randomize() with {nineBits <= 500 && nineBits >= 250;};
```

This constrains the randomize method to produce values for nineBits between 250 and 500 inclusive. There would be no effect on the random number generation for fiveBits. To use the assert statement, you would write:

```
1  assert (tv.randomize() with {nineBits <= 500 && nineBits >= 250;})
2      else $error ("OOPS, randomize didn't work");
```

The comparison in the constraints could be with variables, thus one could write:

```
1  logic [8:0] y;
2  assert (tv.randomize() with {nineBits <= y && nineBits >= 250;})
3      else $error ("OOPS, randomize didn't work");
```

In this case, the value of y can be set by the testbench. Of course, if y is set to a value under 250, then it is impossible to find a random value that satisfies the constraint specified; the randomize method would return FALSE and the $error statement would execute.

The constraints specified in this way could include both of the random variables in the object. The constraints can be any SystemVerilog expression with integral variables or constants, i.e., bit, byte, reg, logic, integer, enum, and packed struct.

Now consider how to generate random inputs for testing the sumItUp hardware thread. This thread, presented in Section 5.1, requires that a packet of information be sent to it. The packet is a series of numbers to add. The first value is on the input when go_l is asserted. The series of numbers is finished when the value 0 is on the input. Thus, 0 cannot be one of the values to add.

Example 7.9 shows two class definitions. The sumPkt class is the constrained random number generator. Class testSumThread generates arrays of random numbers from the sumPkt class to send to the sumItUp hardware thread. It then checks to see that the total value calculated from the values being sent is equal to the sum that the sumItUp thread calculates.

The sumPkt class is defined on lines 1-7 of the example. The howMany random variable will indicate how many values will be in the packet being sent. The dynamic item array (line 3) will hold the sequence of values to be sent. Line 5 specifies a constraint labeled "N." The constraint uses set semantics to specify that the value of howMany must be inside the set specified in the braces "{}". The specification here is that howMany must be in the range of 2-10 inclusive. Thus, even though it is a 16-bit variable, it will only take on values of 2-10.

This set specification could have been more complicated. For instance,

```
constraint N {howMany inside { [2:10], 13, [25:73] }; }
```

would also include 13 and values between 25 and 73 inclusive as values that howMany could take on.

The second constraint, called "arraySize," specifies that item.size, the number of elements in the dynamic array, must be equal to howMany. When randomize is called on an object of type sumPkt, these two constraints will be solved together, producing an array of howMany random numbers in the dynamic array. If the constraint solver is unable to find values that satisfy the constraints — for instance, they might have been miss-specified — then randomize will return FALSE.

Class testSumThread is shown on lines 9-29 of Example7.9. It has a local variable to store the total it calculates; this is used by the checkTotal method which compares it to the value passed to it to determine if the sumItUp hardware thread correctly calculated the total. On line 11, an object of type sumPkt is instantiated with the name pkt.

The sendPktToThread method calls randomize on pkt, producing the dynamic array pkt.item[] with pkt.howMany random values in it. These are then sent to the design-under-test (the sumItUp thread) followed by sending it a 0. While sending, it calculates the total of the values sent.

The sendPktToThread task is called from the initial block shown in Example7.10. This block instantiates an object t of type testSumThread and then enters a loop. The for loop will cycle 100 times, calling the sendPktToThread task. As described above, for each call it generates a packet of information and sends it to the design-under-test. When done is asserted by the design, checkTotal is called with the output value of the design (sum) to check if it is correct. If not, the assert fails.

Thus the for loop sends 100 packets to the thread, with randomly generated length and content.

7.5.3 Random case

The random case (randcase) procedural statement is used to select one of its statements at random. Consider the situation where we want to test a net-

```
1   class sumPkt;
2     rand bit [15:0] howMany;
3     rand bit [15:0] item[];
4
5     constraint N         {howMany inside {[2:10]};}
6     constraint arraySize {item.size == howMany;}
7   endclass
8
9   class testSumThread;
10    local bit unsigned [15:0] total;
11    local sumPkt pkt = new;
12
13    task sendPktToThread;
14      total = 0;
15      assert (pkt.randomize()) else $error ("oops");
16      for (byte i = 0; i < pkt.howMany; i++) begin
17        @(posedge ck);
18        inA <= pkt.item[i];
19        total += pkt.item[i];
20        go_l <= (i == 0) ? 0 : 1;
21      end
22      @(posedge ck);
23      inA <= 0;
24    endtask
25
26    function checkTotal (bit unsigned [15:0] value);
27      return value == total;
28    endfunction
29  endclass
```

Example7.9 — Class Declarations to Generate Random Inputs to the sumItUp Hardware Thread

```
1   initial begin
2     testSumThread t = new;
3     for (int i = 0; i < 100; i++) begin
4       t.sendPktToThread();
5       wait (done);
6       ChkTotal: assert (t.checkTotal(sum))
7         else $error("OOPS");
8     end
9     $finish;
10  end
```

Example7.10 — Testbench Thread That Works with Example7.9

in Figure 2.2. The other case selection items can be paired up with other covered implicants in the Karnaugh map.

Note that in this example, the case selection items on lines 7 and 10 overlap. That is, if {a, b, c} was 3'b101, that would match with either line 7 or line 10. There is nothing wrong with this, especially when you recognize that these two lines represent the two prime implicants where the function is zero, and these two implicants overlap in the Karnaugh map.

In terms of executing the casez statement, the first case selection item that matches is executed. For input 3'b101, this would be 3'b1?? on line 7 because by definition, a case statement (or casez in the example) executes the first case selection item that matches and none other. The case selection item on line 10 would only be executed when the input was 3'b001 even though it also matches 3'b101.

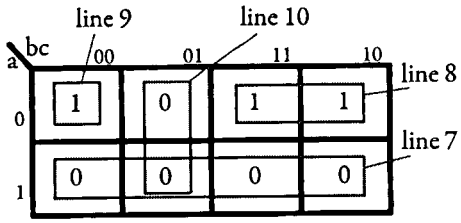

Figure 2.2 — Karnaugh Map

Another caution must be mentioned when using casez. The definition of casez is that a z in any of the bits of the case selection expression is treated as a don't-care in the comparisons with the case selection items. Thus if your design might have high impedance z values due to the use of tri-state drivers, for instance, then the simulation of your casez might not be what you first expect.

2.2.6 Modeling Using Asserted and Not-Asserted Values

To this point, the examples have been fairly lax about naming conventions for logic variables, but in real designs scalar (single bit) signals typically have an assertion level associated with them, indicating which logic value, 0 or 1, indicates whether the signal or condition is asserted. For instance, we could design a sensor for a door to indicate whether the door is open or not. We could design the sensor so that when the door is open, the sensor's output is a logic 1. Or we could design it so that the door being open is indicated by the sensor's output being logic 0. The choice of which way to design the circuit is up to the designer; both are valid choices

The choice is called the *assertion level* for the signal. In the above situation suppose that we want to have a logic variable called doorOpen. But how should we indicate what value, 0 or 1, indicates that the door is open? Designers often add a textual notation to indicate this. For instance, doorOpen_h specifies that a logic 1 indicates that the door is open; the signal is *asserted high*. Alternately, we could have named the signal doorOpen_l if the electronics were designed so that logic 0 indicates that the door is open; the signal is *asserted low*. Again, either specification is valid.

There is no standard that everyone uses for these notations, and often they are just an agreement among the designers on a design team. To indicate that a signal is asserted high, some common notations for variable doorOpen are: doorOpen_h, doorOpen_H, and just doorOpen. This last approach assumes that any scalar without an _h or _H is asserted high. Some common notations for variables asserted low are doorOpen_l, doorOpen_L, doorOpenL, doorOpenB, and doorOpenN.