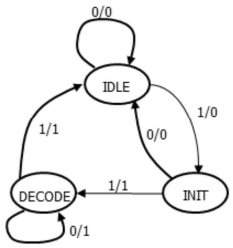
Transition Coverage

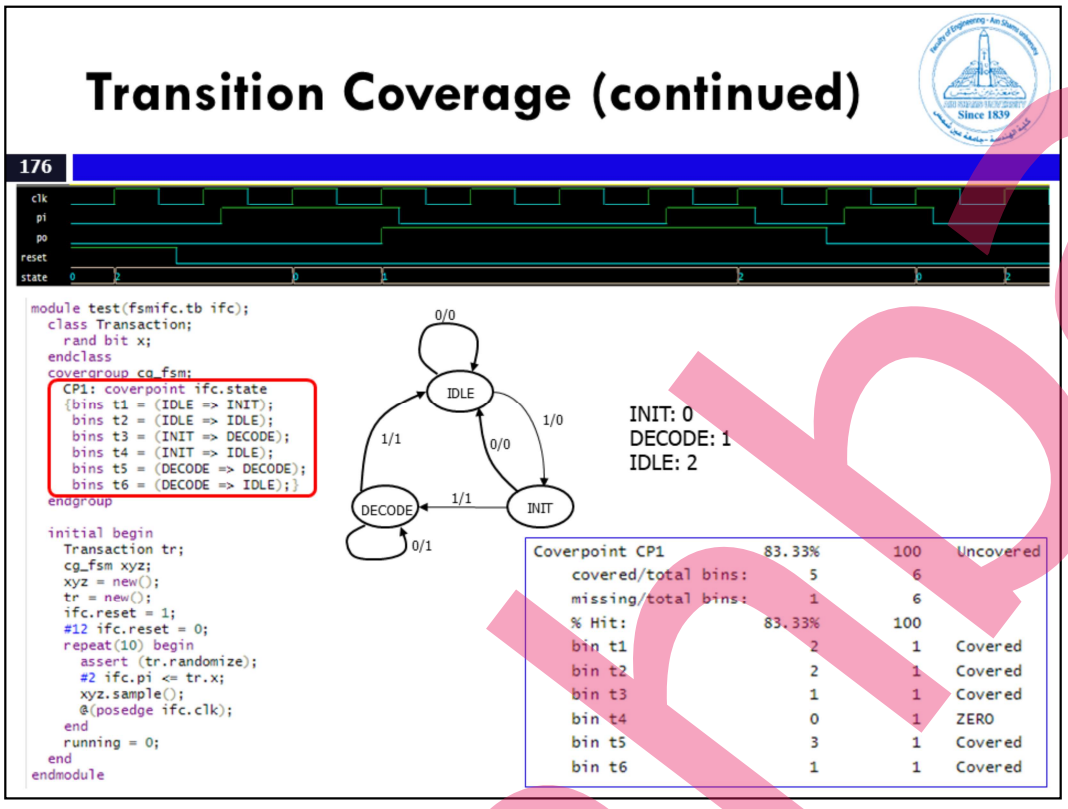

```
bit running = 1;
typedef enum {INIT, DECODE, IDLE} fsmstate_e;

interface fsmifc(input bit clk);
  bit pi;
  bit po;
  bit reset;
  fsmstate_e state;
  modport tb(output pi, reset, clk, input po, state);
  modport dut(input pi, reset, clk, output po, state);
endinterface
```




```
module top;
  bit clk;
  initial begin
    clk = 0;
    while(running == 1)
      #5 clk = ~clk;
  end
  fsmifc u1(clk);
  fsm u2(u1.dut);
  test u3(u1.tb);

  initial begin
    $dumpfile("fsm.vcd");
    $dumpvars(clk,u1);
  end
endmodule
```

```
module fsm(fsmifc.dut abc);
  fsmstate_e pstate,nstate;

  always @(abc.reset) begin
    if (abc.reset == 1) nstate = IDLE;
  end

  always @(posedge abc.clk) begin
    case (pstate)
      IDLE: if (abc.pi == 1) begin
        nstate = INIT; abc.po = 0;
      end
      else begin
        nstate = IDLE;abc.po = 0;
      end

      INIT: if (abc.pi == 1) begin
        nstate = DECODE; abc.po = 1;
      end
      else begin
        nstate = IDLE; abc.po = 0;
      end

      DECODE: if (abc.pi == 1) begin
        nstate = IDLE; abc.po = 1;
      end
      else begin
        nstate = DECODE; abc.po = 1;
      end
    endcase
    pstate = nstate;
    abc.state = pstate;
  end
endmodule
```

- Here we will design a finite state machine, that has three states.
- We need to generate a random input value and we want to make sure that all the transitions have been excited.



- You can specify state transitions for a cover point.
- In this way, you can tell not only what interesting values were seen but also the sequences.
- For example, you can check if he state ever went from INIT to IDLE, or from INIT to DECODE.
- The transition in a cover group is specified as follows: (INIT => IDLE) for example.
- You can quickly specify multiple transitions using ranges: the expression (1,2 => 3,4) creates the four transitions (1=>3), (1=>4), (2=>3), and (2=>4).
- You can specify transitions of any length. Note that you have to sample once for each state in the transition.
  So (0 = > 1 = > 2) is different from (0 = > 1 = > 1 = > 2) or
  (0 = > 1 = > 1 = > 1 = > 2).
- If you need to repeat values, as in the last sequence, you can use the shorthand form: (0 = > 1 [*3] = > 2). This is equivalet to (0 = > 1 = > 1 = > 1 = > 2)
- To repeat the value 1 for 3, 4, or 5 times. use 1 [* 3 : 5] .

Wild Cards in Cover Groups

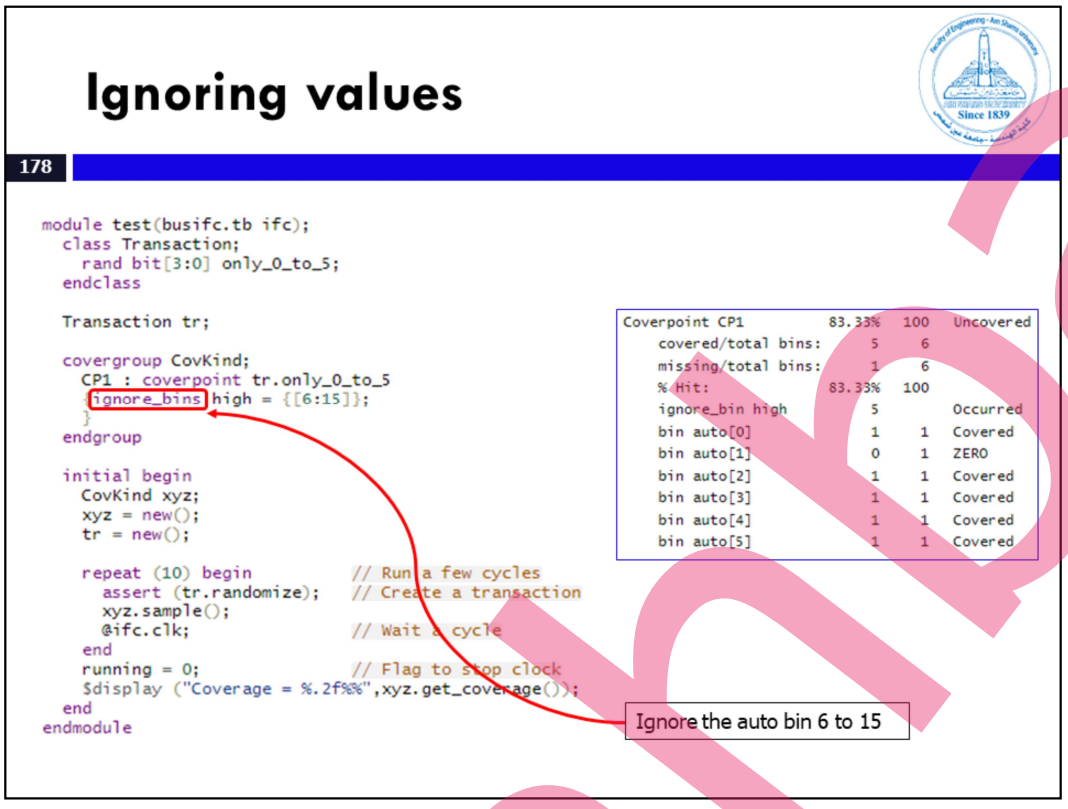
177

```
module test(busifc.tb ifc);
  class Transaction;
    rand bit[3:0] i;
  endclass

  Transaction tr;

  covergroup CovKind;
    CP1 : coverpoint tr.i
    {wildcard bins even = {4'bxzx0};
     wildcard bins odd = {4'b???1};
    }
  endgroup

  initial begin
    CovKind xyz;
    xyz = new();
    tr = new();

    repeat (10) begin          // Run a few cycles
      assert (tr.randomize);   // Create a transaction
      xyz.sample();
      @ifc.clk;                // Wait a cycle
    end
    running = 0;               // Flag to stop clock
    $display ("Coverage = %.2f%%",xyz.get_coverage());
  end
endmodule
```

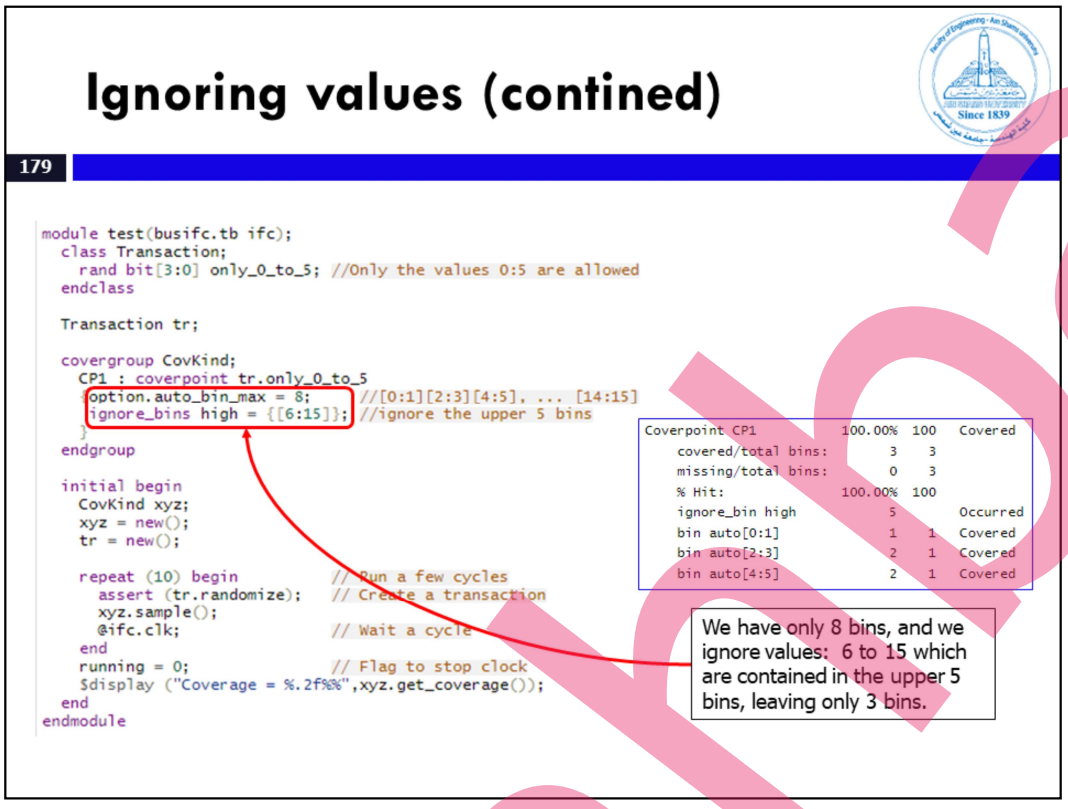
Any x, z, or ? in the expression is treated as a wildcard for 0 or 1

You use the wildcard keyword to create multiple states and transitions

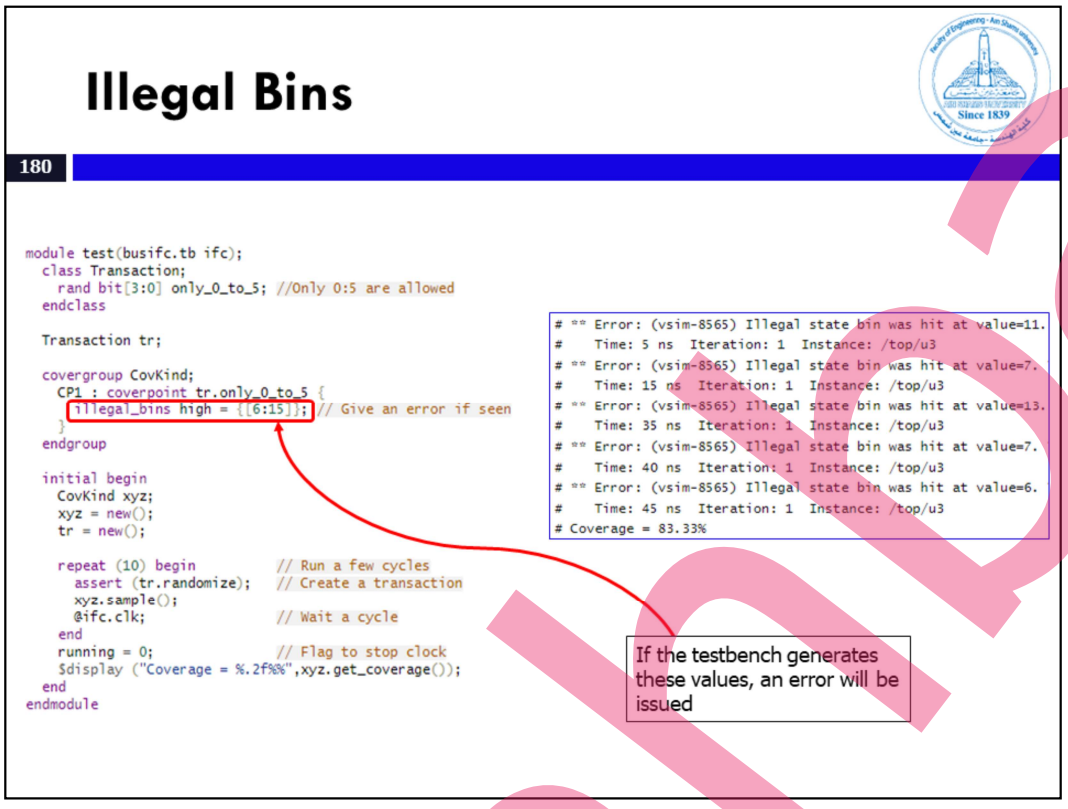
- You use the wildcard keyword to create multiple values for bins.
- Any x, z, or ? in the expression is treated as a wildcard for 0 or 1.
- The shown code creates a cover point with a bin for even values and one for odd values.



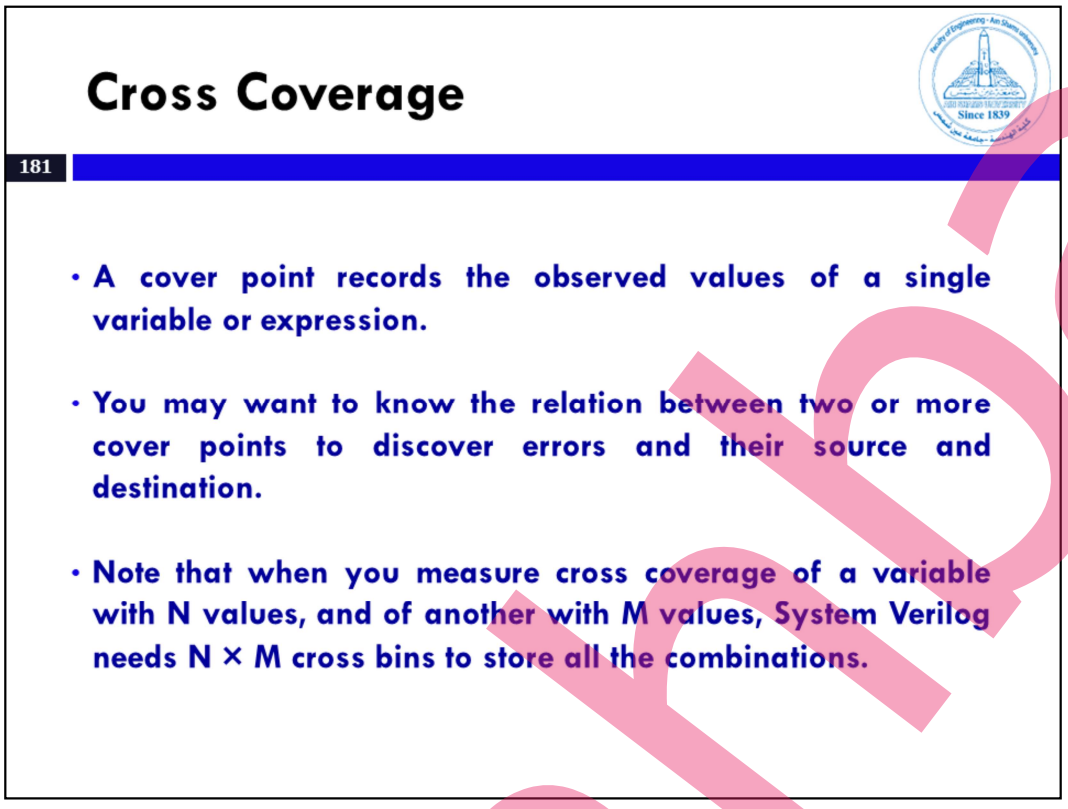
- With some cover points, you never get all possible values. For instance, a 4-bit variable may be used to store just six values, 0-5. If you use automatic bin creation, you never get beyond 37.5% coverage.
- There are two ways to solve this problem:
  - You can explicitly define the bins that you want to cover as shows before
  - Alternatively, you can let SystemVerilog automatically create bins, and then use *ignore_bins* to tell which values to exclude from functional coverage calculation.
- The original range of *only_0_to_5*, a four-bit variable is 0:15. The *ignore_bins* excludes the last 10 bins, which reduces the range to 0:5. So total coverage for this group is the number of bins with samples, divided by the total number of bins, which is 6 in this case.



- If you define bins either explicitly or by using the *auto_bin_max* option, and then ignore them, the ignored bins do not contribute to the calculation of coverage. In the above code sample, eight bins are initially created using the *auto_bin_max* option: [0:1], [2:3], [4:5], [6:7], [8:9], [10:11], [12:13], [14:15],
- However, then the 5 uppermost bins are eliminated by *ignore_bins,* and so at the end only three bins are created. This cover point can have coverage of 0%, 33%, 66%, or 100%.



- Some sampled values not only should be ignored but also should cause an error if they are seen.
- This is best done in the testbench's monitor code, but can also be done by labeling a bin with *illegal_bins*.
- Use *illegal_bins* to catch states that were missed by the test's error checking.
- This also double-checks the accuracy of your bin creation: if an illegal value is found by the cover group, it is a problem either with the testbench or with your bin definitions.



- A cover point records the observed values of a single variable or expression.
- You may want to know not only which transactions have happened, but also which values were generated due to the occurrence of these transactons.
- For this you need cross coverage that measures what values were seen for two or more cover points at the same time.
- Note that when you measure cross coverage of a variable with N values, and of another with M values, System Verilog needs N × M cross bins to store all the combinations.

Basic Cross Coverage Example

182

```
module test(busifc.tb ifc);
  class Transaction;
    rand bit [2:0] p;
    rand bit [3:0] k;
  endclass

  Transaction tr;

  covergroup CovKind;
    CP1 : coverpoint tr.p;
    CP2: coverpoint tr.k;
    cross CP1,CP2;
  endgroup

  initial begin
    CovKind xyz;
    xyz = new();
    tr = new();

    repeat (30) begin
      assert (tr.randomize); // Create a transaction
      xyz.sample();
      @ifc.clk;              // Wait a cycle
    end
    running = 0;             // Flag to stop clock
    $display ("Coverage = %.2f%%",xyz.get_coverage());
  end
endmodule
```

```
                                  Metric       Goal
-------------------------------------------------------
Coverpoint CP1                  100.00%        100
    covered/total bins:               8          8
    missing/total bins:               0          8
    % Hit:                      100.00%        100
Coverpoint CP2                   87.50%        100
    covered/total bins:              14         16
    missing/total bins:               2         16
    % Hit:                       87.50%        100
Cross #cross__0#                 20.31%        100
    covered/total bins:              26        128
    missing/total bins:             102        128
    % Hit:                       20.31%        100
```

```
                                  Metric Goal Bins    Status
-------------------------------------------------------------
bin <auto[12],auto[0]>             3    1    -     Covered
bin <auto[5],auto[0]>              1    1    -     Covered
bin <auto[2],auto[0]>              3    1    -     Covered
bin <auto[0],*>                    0    1    8     ZERO
bin <auto[14],auto[7]>             0    1    1     ZERO
bin <auto[13],auto[7]>             0    1    1     ZERO
bin <auto[12],auto[7]>             0    1    1     ZERO
bin <auto[9],auto[7]>              0    1    1     ZERO
bin <auto[8],auto[7]>              0    1    1     ZERO
bin <auto[7],auto[7]>              0    1    1     ZERO
bin <auto[6],auto[7]>              0    1    1     ZERO
```

Sample of the generated cross bins

- Some previous examples have measured coverage of the transaction *kind*, and *port number*, but what about the two combined?
- Did you try every kind of transaction into every port?
- The *cross* construct in SystemVerilog records the combined values of two or more cover points in a group.
- The *cross* statement takes only cover points or a simple variable name.
- If you want to use expressions, hierarchical names or variables in an object such as *handle.variable*, you must first specify the expression in a cover point with a label and then use the label in the *cross* statement.
- The above code creates cover points for *tr.k* and *tr.p,* then the two points are crossed to show all combinations. System Verilog creates a total of 128 (8 × 16) bins.
- Even a simple cross can result in a very large number of bins.
- A random testbench created 30 transactions and produced the coverage report shown above.
- Note that even though 100% port values were generated, and 87.5% kind values are generated, only 21.31% of the cross combinations were seen.

Labelling Cross Coverage Bins

183

```
module test(busifc.tb ifc);
  class Transaction;
    rand bit [2:0] p;
    rand bit [3:0] k;
  endclass

  Transaction tr;

  covergroup CovKind;
    CP1 : coverpoint tr.p
    {bins port[] = {[0:$]};}

    CP2: coverpoint tr.k{
      bins zero = {0};          // 1 bin for kind = 0
      bins low = {[1:3]};       // 1 bins for values 1:3
      bins high[] = {[8:$]};// 8 separate bins
      bins misc = default;      // 1 bin for the rest
    }

    cross CP1,CP2;
  endgroup

  initial begin
    CovKind xyz;
    xyz = new();
    tr = new();

    repeat (50) begin
      assert (tr.randomize); // Create a transaction
      xyz.sample();
      @ifc.clk;              // Wait a cycle
    end
    running = 0;             // Flag to stop clock
    $display ("Coverage = %.2f%%",xyz.get_coverage());
  end
endmodule
```

	Metric	Goal	Status
bin <port[1],high[15]>	2	1	Covered
bin <port[0],high[15]>	4	1	Covered
bin <port[7],high[14]>	4	1	Covered
bin <port[6],high[14]>	4	1	Covered
bin <port[4],zero>	9	1	Covered
bin <port[3],zero>	4	1	Covered
bin <port[2],zero>	1	1	Covered
bin <port[1],zero>	6	1	Covered
bin <port[0],zero>	6	1	Covered
bin <port[4],high[13]>	0	1	ZERO
bin <port[1],high[10]>	0	1	ZERO
bin <port[0],high[8]>	0	1	ZERO

Sample of the generated cross bins

- If you want more readable cross coverage bin names. you can label the individual cover point bins, and System Verilog will use these names when creating the cross bins.
- If you define bins that contain multiple values, the coverage statistics change.
- In the generated report, the number of cross bins has dropped from 128 to 80. This is because kind has 10 bins: zero, lo, hi[8], hi[9], hi[10], hi[11], hi[12], hi[13], hi[14], hi[15]. The bin *misc* is not used in coverage calculation.

Excluding Cross Coverage Bins

184

```
module test(busifc.tb ifc);
  class Transaction;
    rand bit [2:0] p;
    rand bit [3:0] k;
  endclass
  Transaction tr;

  covergroup CovKind;
    CP1 : coverpoint tr.p
    {bins port[] = {[0:$]};}

    CP2: coverpoint tr.k{
      bins zero = {0};       // 1 bin for kind = 0
      bins low = {[1:3]};    // 1 bins for values 1:3
      bins high[] = {[8:$]};// 8 separate bins
      bins misc = default;  // 1 bin for the rest
    }

    cross CP1,CP2{
      ignore_bins t1 = binsof(CP1) intersect {7};
      ignore_bins t2 = binsof(CP1) intersect {0} &&
                       binsof(CP2) intersect {[9:11]};
      ignore_bins t3 = binsof(CP2.low);
    }
  endgroup

  initial begin
    CovKind xyz;
    xyz = new();
    tr = new();
    repeat (50) begin
      assert (tr.randomize); // Create a transaction
      xyz.sample();
      @ifc.clk;              // Wait a cycle
    end
    running = 0;             // Flag to stop clock
    $display ("Coverage = %.2f%%",xyz.get_coverage());
  end
endmodule
```

CP2

	zero	lo	hi[8]	hi[9]	hi[10]	hi[11]	hi[12]	hi[13]	hi[14]	hi[15]
port[0]	✓	×	✓	×	×	×	✓	✓	✓	✓
port[1]	✓	×	✓	✓	✓	✓	✓	✓	✓	✓
port[2]	✓	×	✓	✓	✓	✓	✓	✓	✓	✓
port[3]	✓	×	✓	✓	✓	✓	✓	✓	✓	✓
port[4]	✓	×	✓	✓	✓	✓	✓	✓	✓	✓
port[5]	✓	×	✓	✓	✓	✓	✓	✓	✓	✓
port[6]	✓	×	✓	✓	✓	✓	✓	✓	✓	✓
port[7]	×	×	×	×	×	×	×	×	×	×

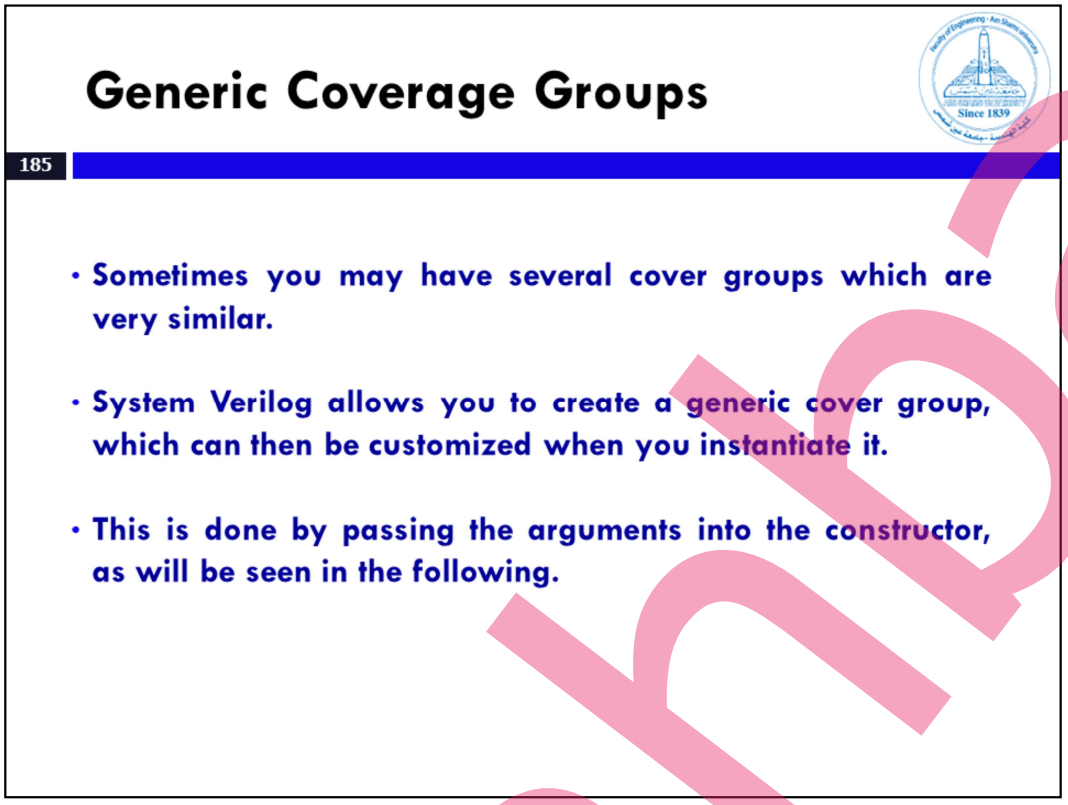
CP1

Ignore the following cross bins:
- bins with port = 7 (10 bins)
- bins with port =0, and kind = 9, 10, 11 (3 bin)
- Bins with port = any value, kind = lo (8 bins)

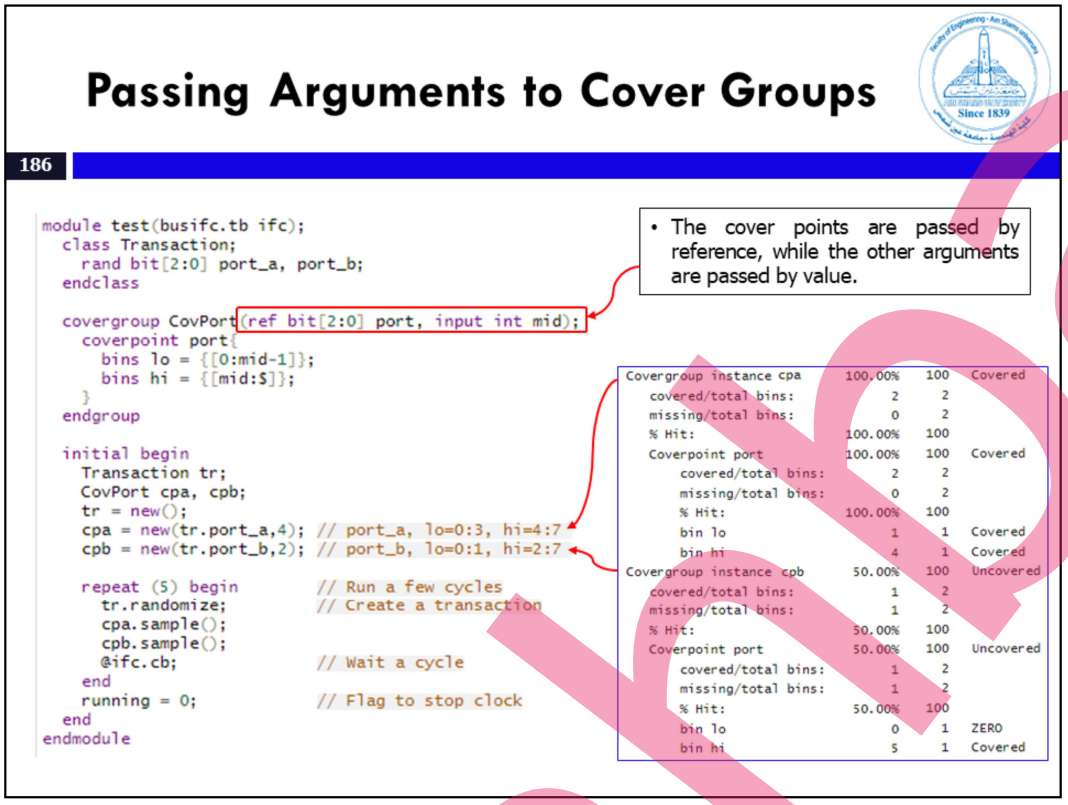
Cross bins used for coverage = 60

```
#  ------------------------------------------------------------------
#  Covergroup                          Metric       Goal   Status
#  ------------------------------------------------------------------
#      Cross #cross__0#                30.00%        100   Uncovered
#          covered/total bins:            18         60
#          missing/total bins:            42         60
#          % Hit:                      30.00%        100
```

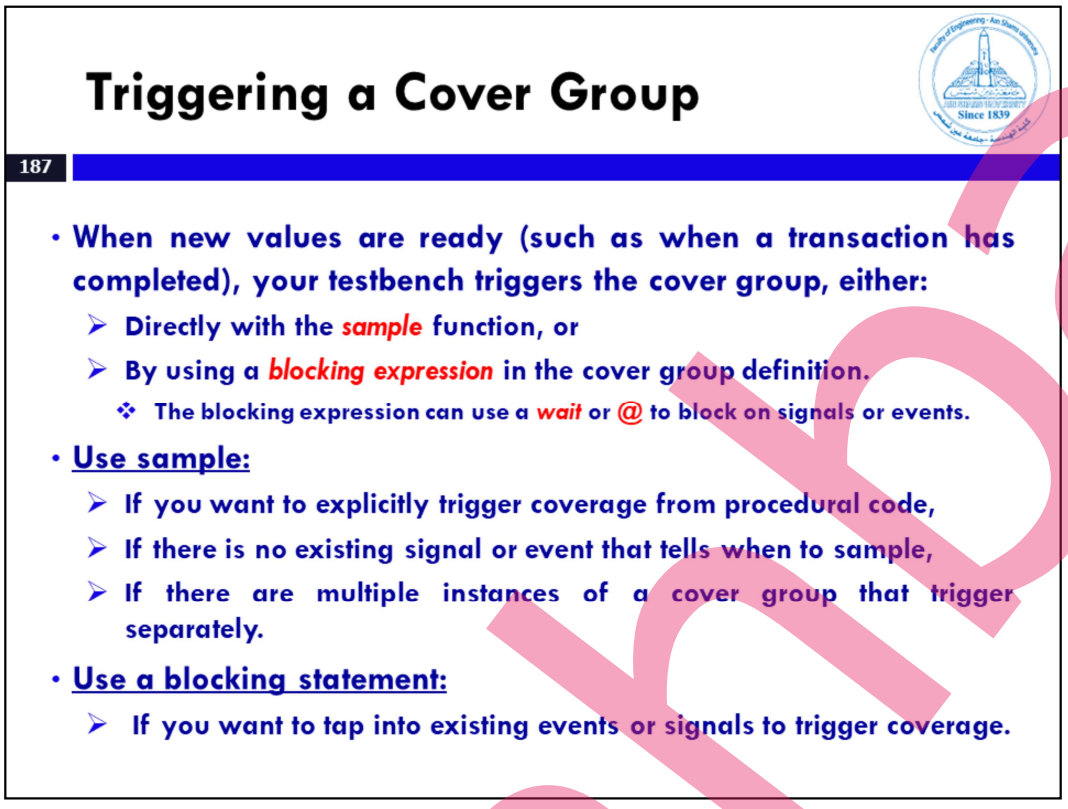
- To reduce the number of bins, use *ignore_bins*. With cross coverage, you specify the cover point with *binsof* and the set of values with *intersect* so that a single *ignore_bins* construct can sweep out many individual bins.
- The first *ignore_bins* (t1) just excludes bins where *port* is 7 and any value of *kind*. Since *kind* has 10 bins, this statement excludes 10 bins.
- The second *ignore_bins* (t2) is more selective, ignoring bins where port is 0 and kind is 9, 10, or 11, for a total of 3 bins.
- The *ignore_bins* can use the bins defined in the individual cover points.
- The *ignore_bins* (t3) uses bin names to exclude *CP2.low*
- The bins must be names defined at compile-time, such as zero and low.
- The bins hi[11], hi[9], hi[10] ... hi[15], and any automatically generated bins do not have names that can be used at compile-time in other statements such as *ignore_bins*; these names are created at run-time or during the report generation.
- Note that *binsof* uses parentheses (), while *intersect* specifies a range and therefore uses curly braces {}



- As you start writing cover groups, you will find that some are very similar to one another.
- System Verilog allows you to create a generic cover group so that you can specify a few unique details when you instantiate it.



- The above code shows a cover group that uses an argument to split the range into two halves. Just pass the midpoint value to the cover groups' new function.
- The cover points are passed by reference, while the other arguments are passed by value.



- The two major parts of functional coverage are the sampled data values and the time when they are sampled.
- When new values are ready (such as when a transaction has completed), your testbench triggers the cover group. This can be done directly with the *sample* function, as shown in the previous examples, or by using a *blocking expression* in the cover group definition (as will be seen in the next few slide).
- The blocking expression can use a *wait* or @ to block on signals or events.
- Use sample if you want to explicitly trigger coverage from procedural code, if there is no existing signal or event that tells when to sample, or if there are multiple instances of a cover group that trigger separately.
- Use the blocking statement in the cover group declaration if you want to tap into existing events or signals to trigger coverage.

Cover Group with an Event Trigger

188

```
module test(busifc.tb ifc);
  class Transaction;
    rand bit [2:0] p;        //Eight port numbers
    rand bit [3:0] k;
  endclass

  Transaction tr;            // Create a handle

  covergroup CovKind @ifc.clk;
    CP1 : coverpoint tr.k;
  endgroup

  initial begin
    CovKind xyz;
    xyz = new();
    tr = new();

    repeat (20) begin          // Run some cycles
      assert (tr.randomize);   // Create a transaction
//      xyz.sample();
      @ifc.clk;                // Wait a cycle
    end
    running = 0;               // Flag to stop clock
    $display ("Coverage = %.2f%%",xyz.get_coverage());

  end
endmodule
```

Here the sampling occurs based on the clock edge of the clocking block.

Here we don't use sample

- In the above code the cover group CovPort is sampled when the testbench triggers the *ifc.clk* event (i.e at the positive edge of the clock)
- The advantage of using an event over calling the sample method directly is that you may be able to use an existing event (even if this event is triggered by an assertion)

Triggering on a SV Assertion

189

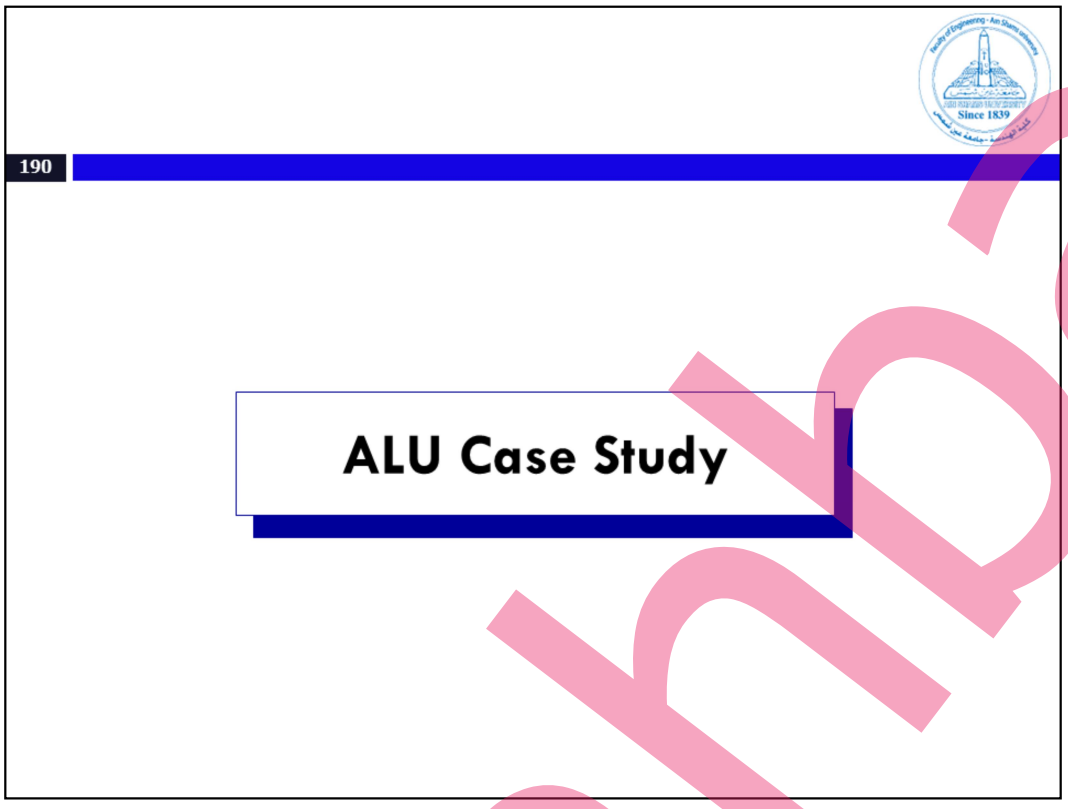
```
module test(busifc.tb ifc);
  event ready;
  class Transaction;
    rand bit [2:0] p;        //Eight port numbers
    rand bit [3:0] k;
  endclass

  Transaction tr;            // Create a handle

  covergroup CovKind @(ready);
    CP1 : coverpoint tr.k;
  endgroup

  initial begin
    CovKind xyz;
    xyz = new();
    tr = new();

    repeat (20) begin          // Run some cycles
      assert (tr.randomize) -> ready;
//      xyz.sample();
//      @(posedge ifc.clk);     // Wait a cycle
    end
    running = 0;               // Flag to stop clock
    $display ("Coverage = %.2f%%",xyz.get_coverage());
  end
endmodule
```

Ready is an event that will be triggered by an assertion

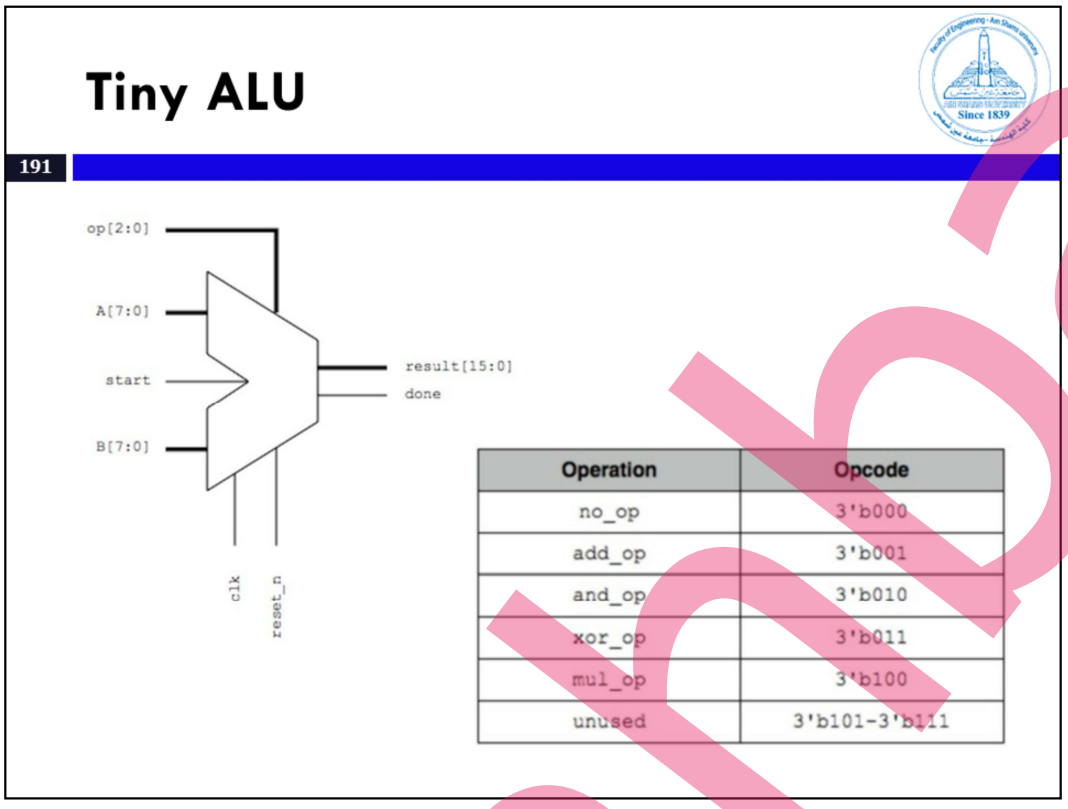
Here the sampling occurs based on the event triggered by the assertion.

The assertion triggers the event when the randomization succeeds

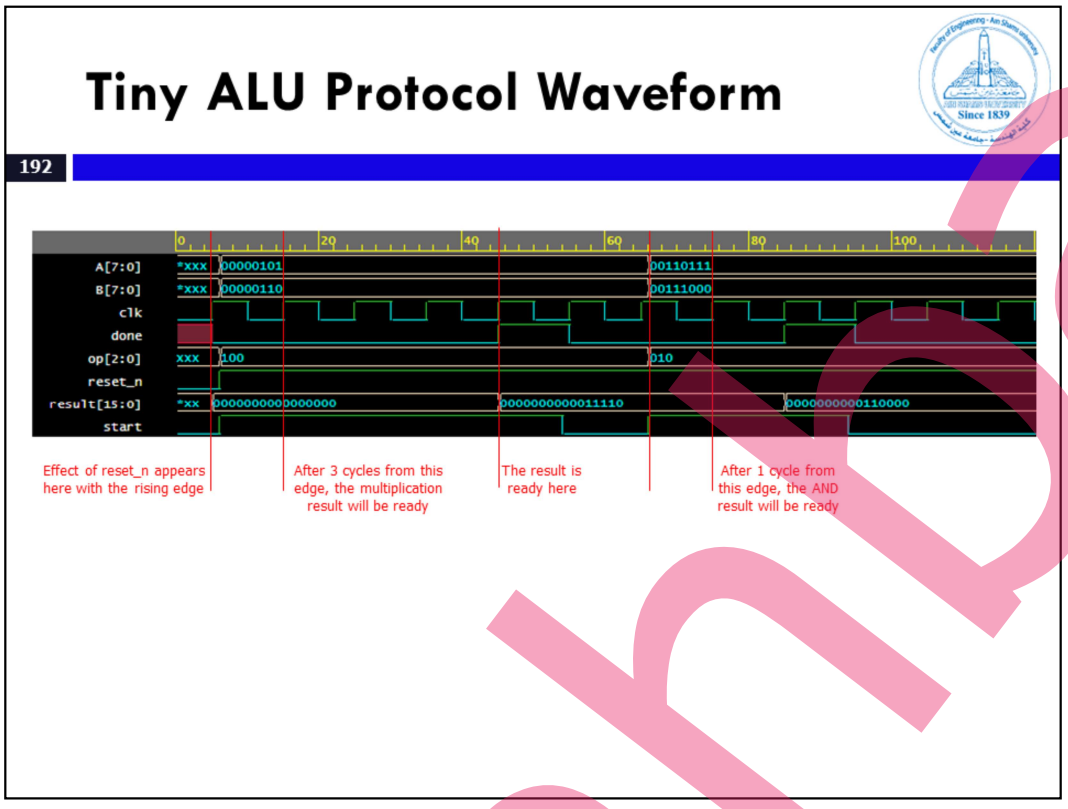
- If you already have an SVA that looks for useful events like a complete transaction, you can add an event trigger to wake up the cover group.
- An identifier declared as an event data type is called a named event.
- Named event is a data type which has no storage.
- A named event can be triggered explicitly using "->".
- Event triggering occurrence can be recognized by using the event control "@".



- Ver



- We will examine the UVM by verifying a simple design: the TinyALU.
- The TinyALU is a simple ALU. It accepts two eight-bit numbers (A and B) and produces a 16-bit result.
- The ALU works at the rising edge of the *clock*. When the *start* signal is active, the TinyALU reads operands of the *A* and *B* busses and an operation of the *op* bus, and delivers the result based on the operation.
- Operations can take any number of cycles. The TinyALU raises the *done* signal when the operation is complete.
- The *reset_n* signal is an active-low, synchronous reset.
- The TinyALU has five operations: NOP, ADD, AND, XOR, and MULT. The user encodes the operations on the three bit *op* bus when requesting a calculation.



- The start signal must remain high, and the operator and operands must remain stable until the TinyALU raises the done signal.
- The done signal stays high for only one clock.
- We'll start our journey through the UVM by creating a conventional testbench of the TinyALU. Then we will modify the testbench to make it UVM compliant. We'll discuss the advantages of using the UVM as we transform the testbench.

Simple testbench with interface (The interface)

193

```
interface aluifc(input bit clk);
  logic[7:0] A,B;
  operation_t op;
  logic reset_n, start, done;
  logic[15:0] result;
  modport tb(output A,B,op,reset_n,start, input done, result,clk);
  modport dut(input A,B,op,reset_n,start,clk, output done, result);
endinterface
```

- The interface between the DUT and the testbench has the following ports:
  - The two 8-bit input operands *A, B*
  - A 3-bit operation code *op*. for the ease of use an new enumerated type is defined to give names for the different opcodes.
  - A reset input that is active low *reset_n*
  - A start signal that must be 1 during the operation till the result is issues, *start*
  - A done signal that is set to 1 for one clock cycle when the operation ends. *done*
  - A 16-bit result, *result*
- The clock is an input to the interface as it comes from an external source.
- Modports, are used to determine the direction of these ports when used at the DUT side and at the testbench side.
- Note that *clk* is input to both the DUT and the testbench.

Simple testbench with interface (The testbench)

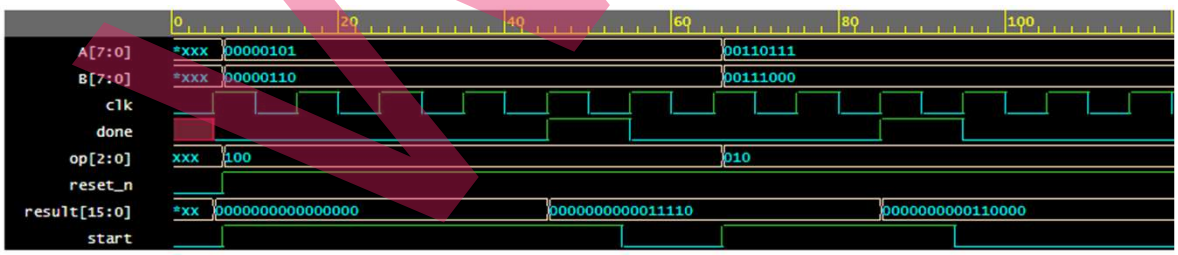
194

```
38 module test(aluifc.tb y);
39   initial begin
40     y.reset_n = 0;
41     y.start = 0;
42
43     # 6;
44     y.reset_n = 1;
45     y.A = 8'b00000101;
46     y.B = 8'b00000110;
47     y.start = 1;
48     y.op = mul_op;      // Multiply operation
49
50     #60
51     y.A = 8'b00110111;
52     y.B = 8'b00111000;
53     y.start = 1;
54     y.op = and_op;      // AND operation
55
56     #30      // just delay to see a complete waveform
57
58     running = 0;
59   end
60
61   always @(posedge y.done) #9 y.start = 0;
62 endmodule
```

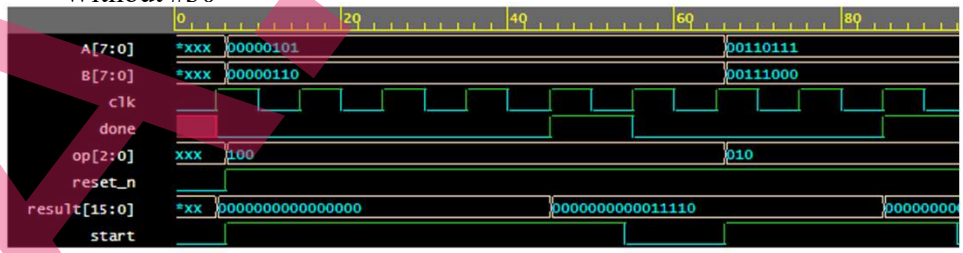
- This is a very simple testbench driving the values of some inputs to the DUT just to visualize some waveforms to make a fast preliminary check.
- **Practical note:**

  The simulator stops with the last event happening on any signal in the waveform. So you will not see the last change in the wave diagram. So we add some delay (#30), just to continue displaying the last part of the wave diagram.

With #30



Without #30



Simple testbench with interface (The ALU)


```
bit running = 1;
typedef enum bit[2:0] {
          no_op  = 3'b000,
          add_op = 3'b001,
          and_op = 3'b010,
          xor_op = 3'b011,
          mul_op = 3'b100} operation_t;

module alu(aluifc.dut x);
  always @(posedge x.clk)
  begin
    if (x.reset_n == 0) begin
      x.result = 16'b0000000000000000;
      x.done = 1'b0;
    end
  end
```

```
  always @ (posedge x.clk)
  begin
    if (x.reset_n != 1'b0) begin
      if (x.start == 1'b1)
        case(x.op)
          no_op: x.done = 1'b1;
          add_op: begin
            @(posedge x.clk) x.result = x.A + x.B;
                    x.done = 1'b1;
                  end
          and_op: begin
            @(posedge x.clk) x.result = x.A & x.B;
                    x.done = 1'b1;
                  end
          xor_op: begin
            @(posedge x.clk) x.result = x.A ^ x.B;
                    x.done = 1'b1;
                  end
          mul_op: begin
            repeat(3) @(posedge x.clk);
                    x.result = x.A * x.B;
                    x.done = 1'b1;
                  end
          default: ;
        endcase
    end
  end
endmodule
```

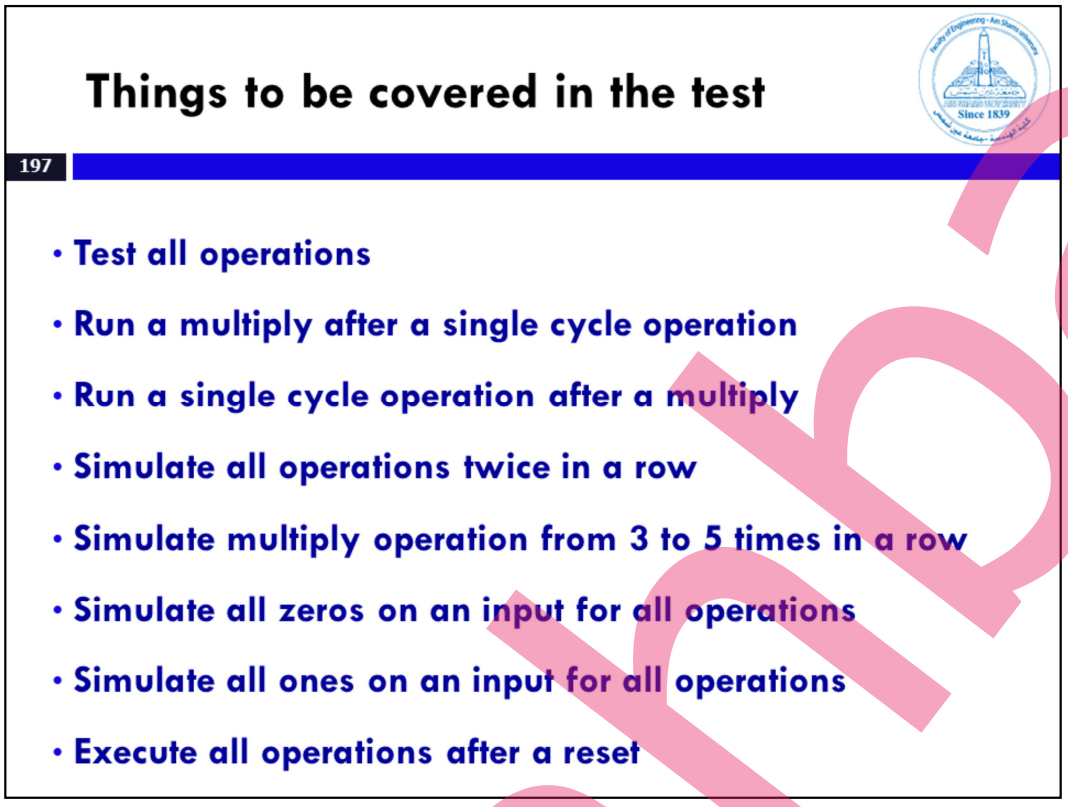
- Here is the code for the ALU to be tested.
- We defined an enumerated type called operation_t to facilitate expressing the different opcodes.

Simple testbench with interface (The top module)

196

```
13 module top;
14   bit clk;
15   initial begin
16     clk = 0;
17     while(running == 1)
18       #5 clk = ~clk;
19   end
20
21   aluifc u1(clk);
22   alu u2(u1.dut);
23   test u3(u1.tb);
24
25   initial begin
26     $dumpfile("alu.vcd");
27     $dumpvars;
28   end
29 endmodule
```

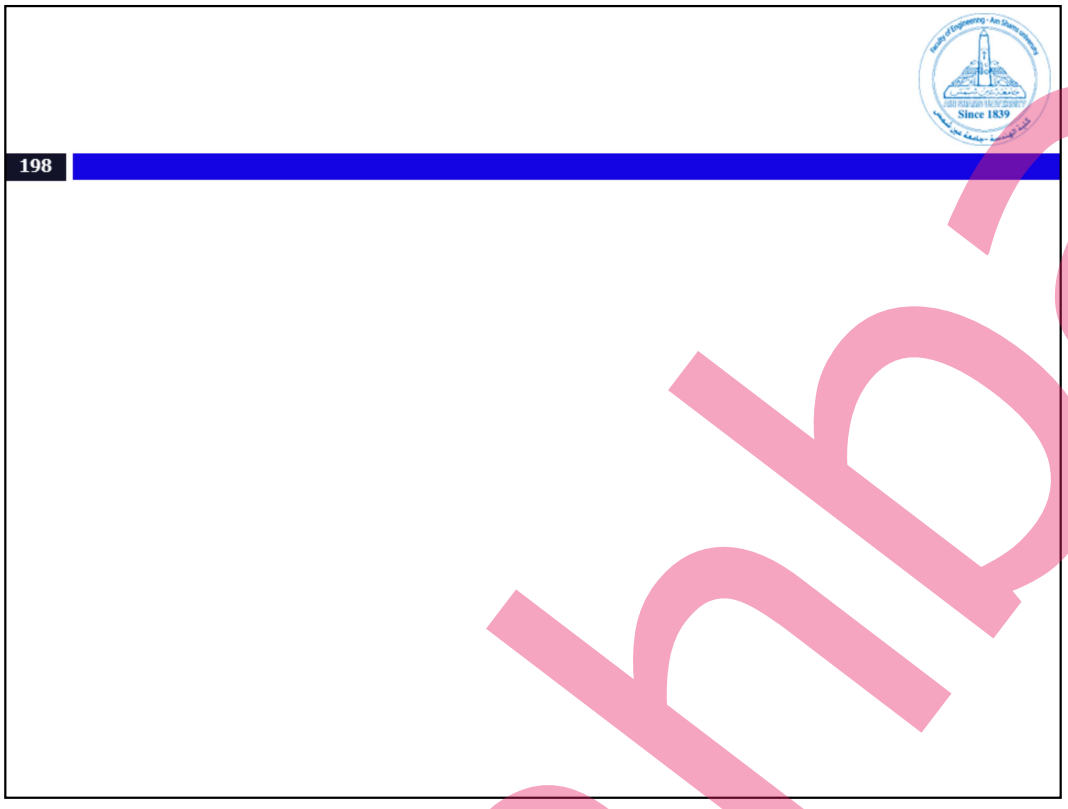
- Here is the top module.
- It has a clock generator that works as long as the global variable *running* = 1
- It also has an initial block that specifies the file name where the wave diagram info is to be stored.



- Once we have made a preliminary test, in the previous slides, let's now revert to random testing. In this case we have to measure the coverage to know when to stop.
- We'll use SystemVerilog cover groups to implement the ALU coverage model.
- Here are the coverage goals:
  - Test all operations
  - Simulate all zeros on an input for all operations
  - Simulate all ones on an input for all operations
  - Execute all operations after a reset
  - Run a multiply after a single cycle operation
  - Run a single cycle operation after a multiply
  - Simulate all operations twice in a row

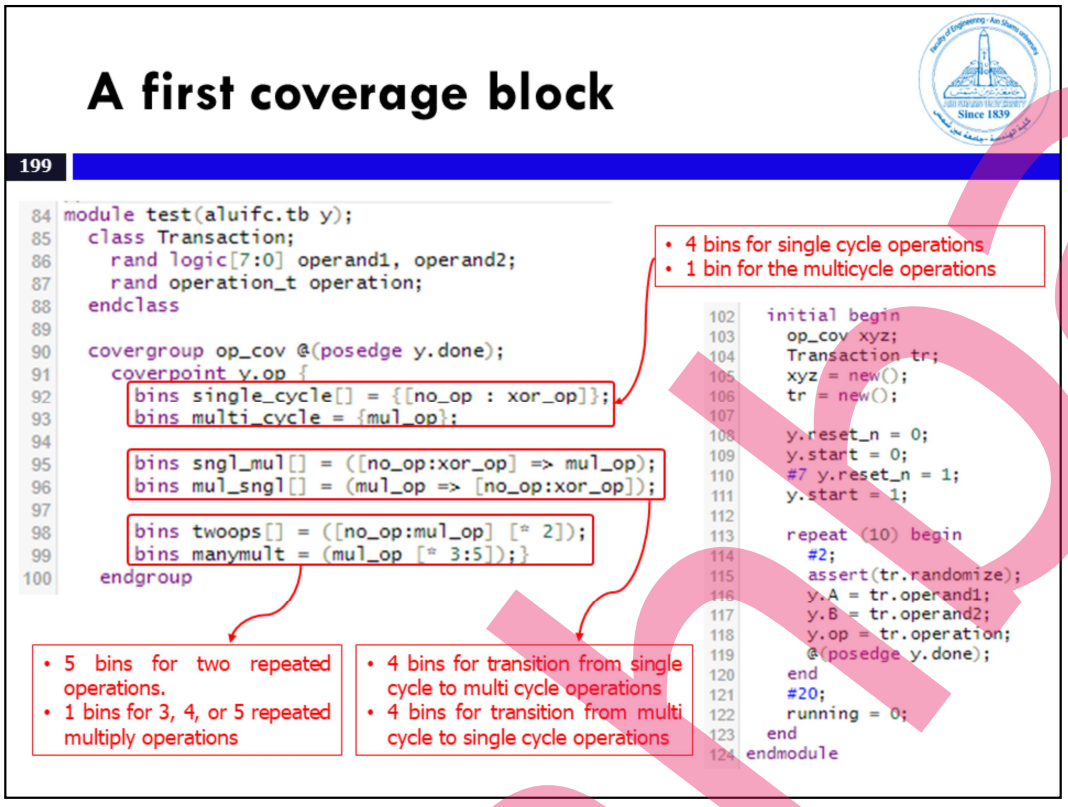
  We can be sure that our ALU is working if we've tested all these scenarios and not gotten any errors. We'll also check that we've got 100% code coverage.

  In the next few slides we will show how we will add the cover groups in our testbench.

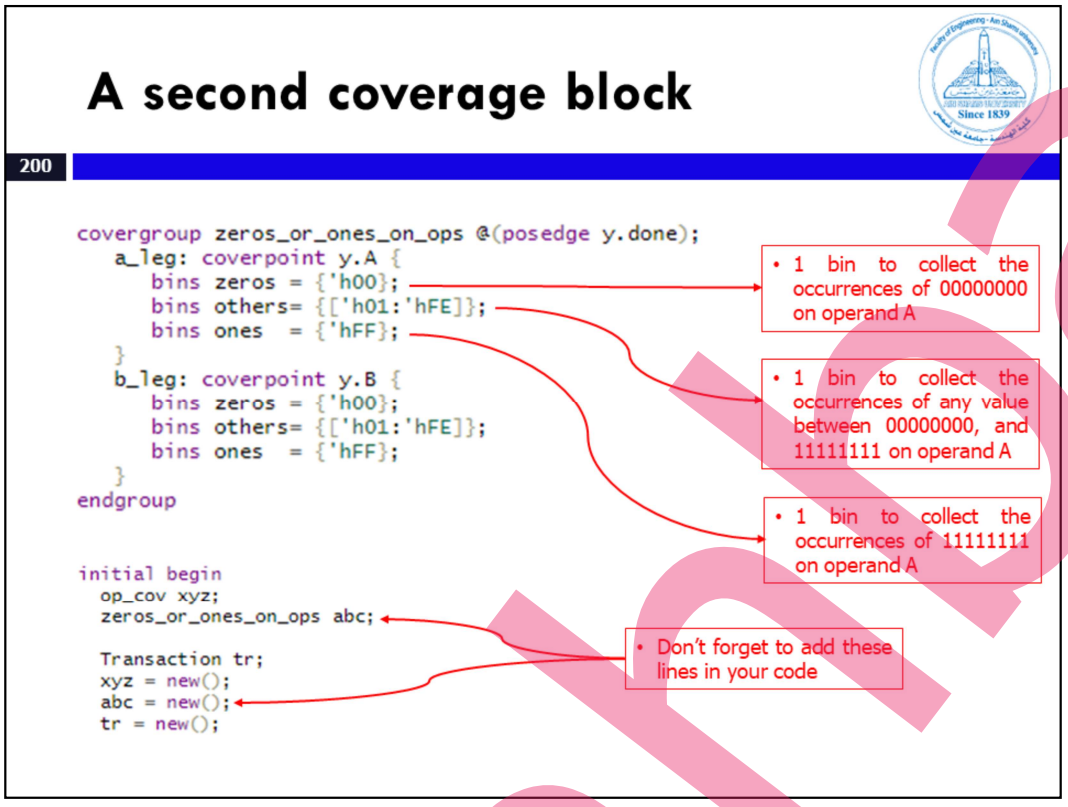


```
bit running = 1;
typedef enum bit[2:0] {no_op  = 3'b000,
         add_op = 3'b001,
         and_op = 3'b010,
         xor_op = 3'b011,
         mul_op = 3'b100} operation_t;

//-------------------------------------------------
interface aluifc(input bit clk);
  logic[7:0] A,B;
  operation_t op;
  logic reset_n, start, done;
  logic[15:0] result;
  modport tb(output A,B,op,reset_n,start, input done, result,clk);
  modport dut(input A,B,op,reset_n,start,clk, output done, result);
endinterface

//-------------------------------------------------
module top;
  bit clk;
  initial begin
    clk = 0;
    while(running == 1)
      #5 clk = ~clk;
  end

  aluifc u1(clk);
  alu u2(u1.dut);
  test u3(u1.tb);

  initial begin
    $dumpfile("alu.vcd");
    $dumpvars;
  end
endmodule
```

```
module alu(aluifc.dut x);
  always @(posedge x.clk)
  begin
    if (x.reset_n == 0) begin
      x.result = 16'b0000000000000000;
      x.done = 1'b0;
    end
  end

  always @ (posedge x.clk)
  begin
    if (x.done == 1) begin
      x.done = 1'b0;
    end
  end

  always @ (posedge x.clk)
  begin
    if (x.reset_n != 1'b0) begin
      if (x.start == 1'b1)
        case(x.op)
          no_op: x.done = 1'b1;
          add_op: begin
            @(posedge x.clk) x.result = x.A + x.B;
                    x.done = 1'b1;
                  end
          and_op: begin
            @(posedge x.clk) x.result = x.A & x.B;
                    x.done = 1'b1;
                  end
          xor_op: begin
            @(posedge x.clk) x.result = x.A ^ x.B;
                    x.done = 1'b1;
                  end
          mul_op: begin
            repeat(3) @(posedge x.clk);
                    x.result = x.A * x.B;
                    x.done = 1'b1;
                  end
          default: ;
        endcase
    end
  end
endmodule
```

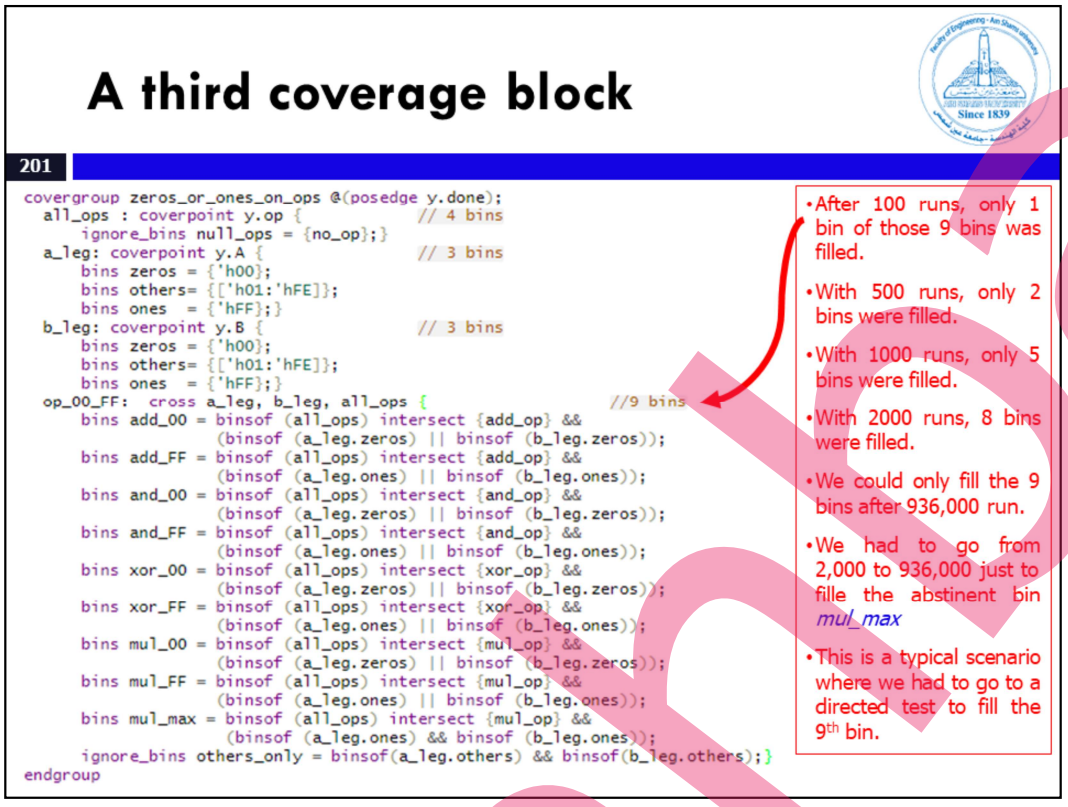


- We want to generate random values for the operands and the opcode to try different scenarios of operation.
- The easiest way to generate random values is as shown above.
  - Create a class and put the keyword *rand* before the variables you need to generate random values for.
  - You then create an object of that class, and initialize it using the constructor *new()*.
  - When you want to get a random value, use *assert(object_name.randomize)*.
- In the above code we measure the coverage for the first 5 items of the previous slide. We will later add extra code to cover all the points.
- We'll use cover groups to capture functional coverage. We declare the cover groups, instantiate them, and use them for sampling.
- First, let's look at the definitions:
  - These definitions show some of the bins in the coverage model.
  - The ***op_cov*** covergroup makes sure that we've covered all the operations and the possible interactions between them.
- We sample our cover points at the rising edge of the *done* signal, when the result of the operation is ready.
- To jump to the end of the story, this testbench achieves 100% functional coverage:



- Add a coverage group as the shown above to your code.
- This group will create 3 bins for the operand A, and 3 bins for the operand B.
- It will ensure the coverage of the cases, where the corner values "00000000" and "11111111" have been tried on operand A, and operand B.
- Later we will make sure that these values have been tried with all operations.
- First, let's look at the definitions:
- These definitions show some of the bins in the coverage model.
- The ***zeros_or_ones_on_ops*** covergroup checks to see if we've had all zeros and all ones on the data ports.
- Once we've defined the covergroups we need to declare, instantiate, and sample them.
- If you don't get 100% coverage you may need to increase the number of runs.

A third coverage block



- We will add some code to the coverage group *zeros_or_ones_on_ops*, and perform cross coverage.
- *all_ops* will automatically generate 4 bins.
- *op_00_FF*: wiil check cross coverage of things that should happen together. (9 bins out of 36 possible bins from cross interaction 3 × 3 × 4)
  - 1. add_00: is a bin to collect the occurrences of an add operation, while at least one of its operands is 00000000.
  - 2. add_FF: is a bin to collect the occurrences of an add operation, while at least one of its operands is 11111111.
  - 3. and_00: is a bin to collect the occurrences of an and operation, while at least one of its operands is 00000000.
  - 4. and_FF: is a bin to collect the occurrences of an and operation, while at least one of its operands is 11111111.
  - 5. xor_00: is a bin to collect the occurrences of an xor operation, while at least one of its operands is 00000000.
  - 6. xor_FF: is a bin to collect the occurrences of an xor operation, while at least one of its operands is 11111111.
  - 7. mul_00: is a bin to collect the occurrences of an mul operation, while at least one of its operands is 00000000.
  - 8. mul_FF: is a bin to collect the occurrences of an mul operation, while at least one of its operands is 11111111.
  - 9. mul_max: is a bin to collect the occurrences of an mul operation, while both of its operands is 11111111.

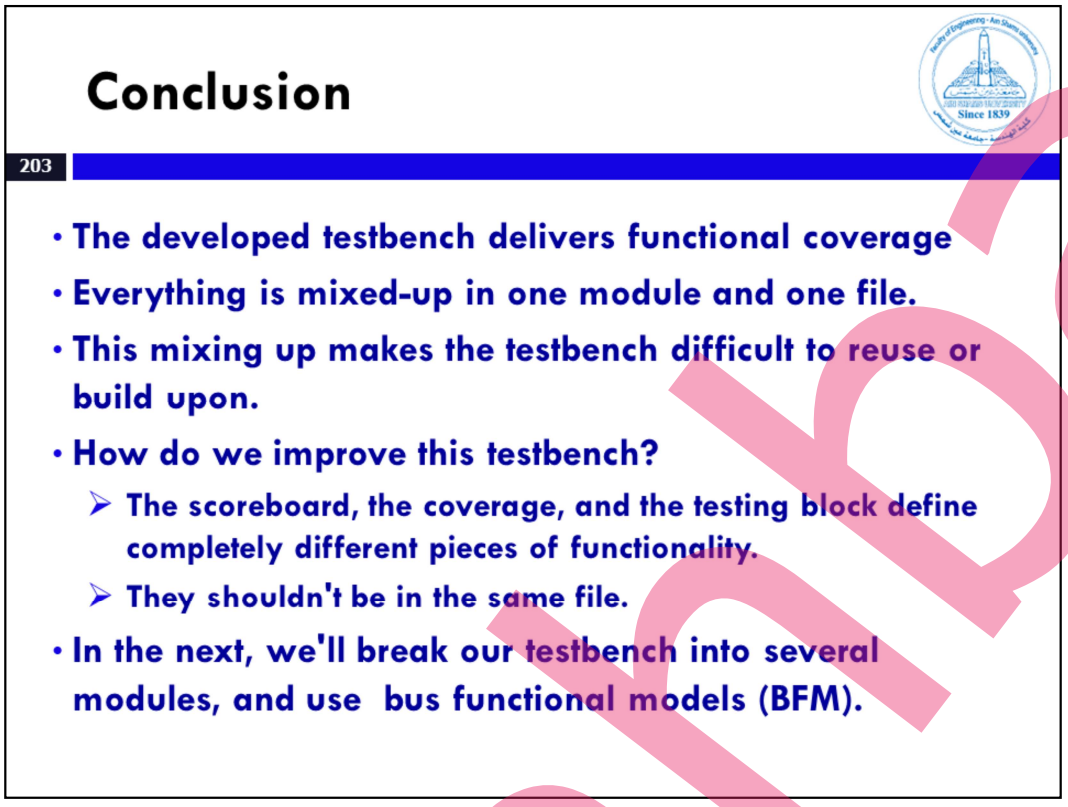
- The 9th bin is very hard to fill using random testing. With only 2000 random runs, we could fill 8 out of the available 9 bins.
- To fill the 9th bin we had to run about 1 million random runs.
- This is a typical scenario where we run random tests for a while, and then for the unfilled bins we create special tests to achieve 100% functional coverage.
- Now imagine how difficult it would be to modify this testbench to create new tests. For one thing, we'd have to completely rewrite this block. That's why UVM is invented as we will see later.

Self checking with the scoreboard block

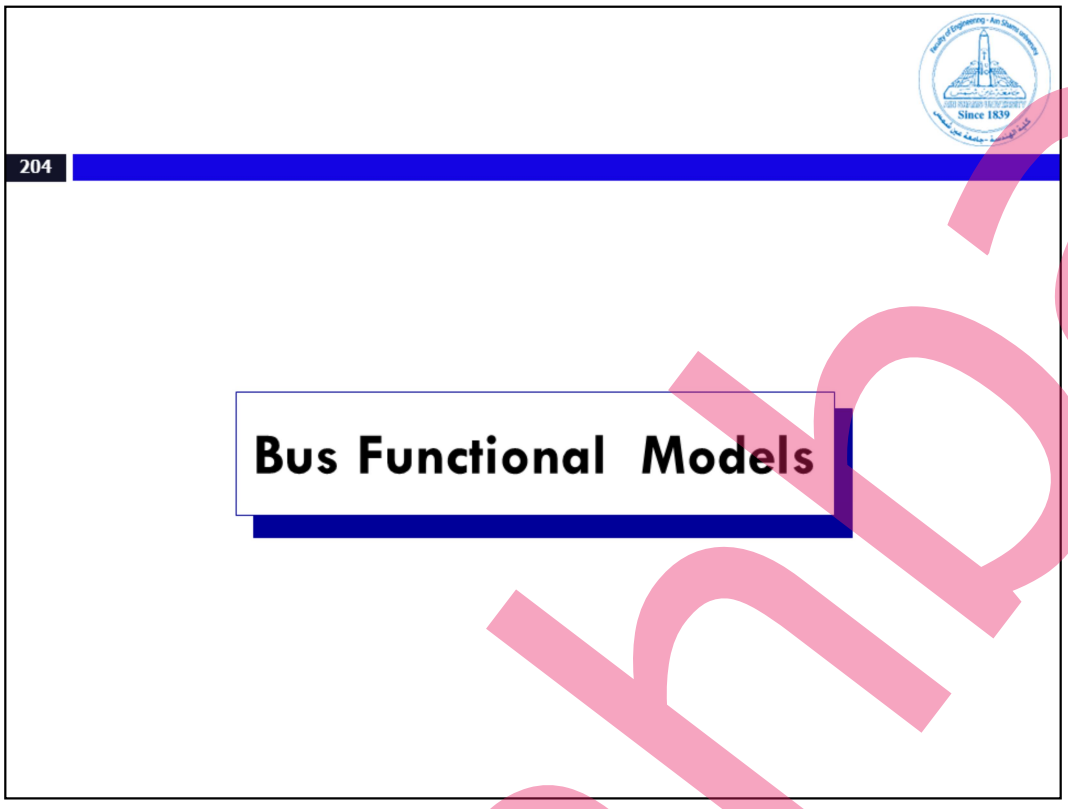
202

```
always @(posedge y.done) begin : scoreboard
  shortint predicted_result;
  #1;
  case (y.op)
    add_op: predicted_result = y.A + y.B;
    and_op: predicted_result = y.A & y.B;
    xor_op: predicted_result = y.A ^ y.B;
    mul_op: predicted_result = y.A * y.B;
  endcase

  if (y.op != no_op)
    if (predicted_result != y.result)
      $error("FAILED: A: %0d  B: %0d  op: %s result: %0d",
             y.A,y.B,y.op.name(),y.result);
end : scoreboard
```

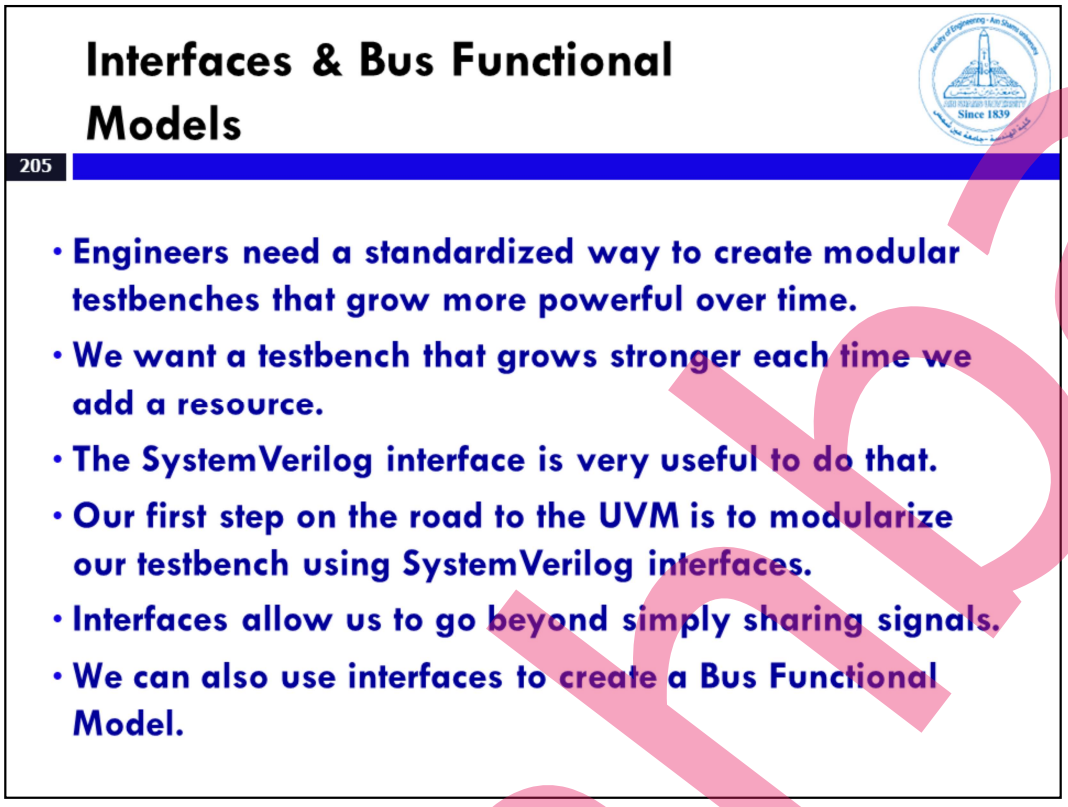
- The scoreboard block checks the actual ALU results against predicted results. The always block watches the *done* signal. When the *done* signal goes high, the scoreboard predicts the ALU output based on the inputs and checks that the output is correct.
- Till now, we have successfully developed the ALU testbench.



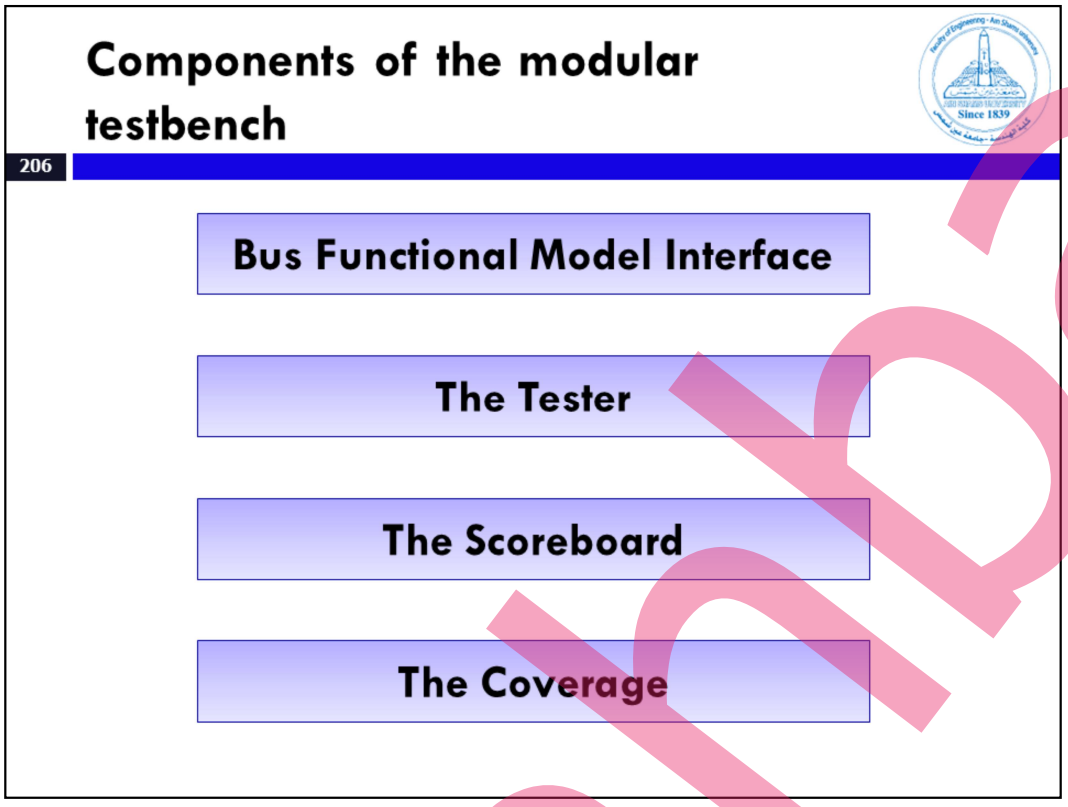
- Our testbench delivers functional coverage, checks that all the operations work properly, and required very little work in stimulus generation.
- It delivered all of this in *one module* and *one file*. Yet this is a poor example of a testbench. All the behavior is mixed up together in one file.This mixing up makes the testbench difficult to reuse or build upon.
- How do we improve this testbench? The first thing we notice is that the scoreboard block, the coverage block, and the testing block each define completely different pieces of functionality.
- They shouldn't be in the same file. By separating them, we'll be able to reuse portions of the testbench in other testbenches and easily modify parts of the testbench such as the stimulus.
- In our next part of the course, we'll break our testbench up into modules and we'll learn about how to use a SystemVerilog Interface as a bus functional model (BFM).



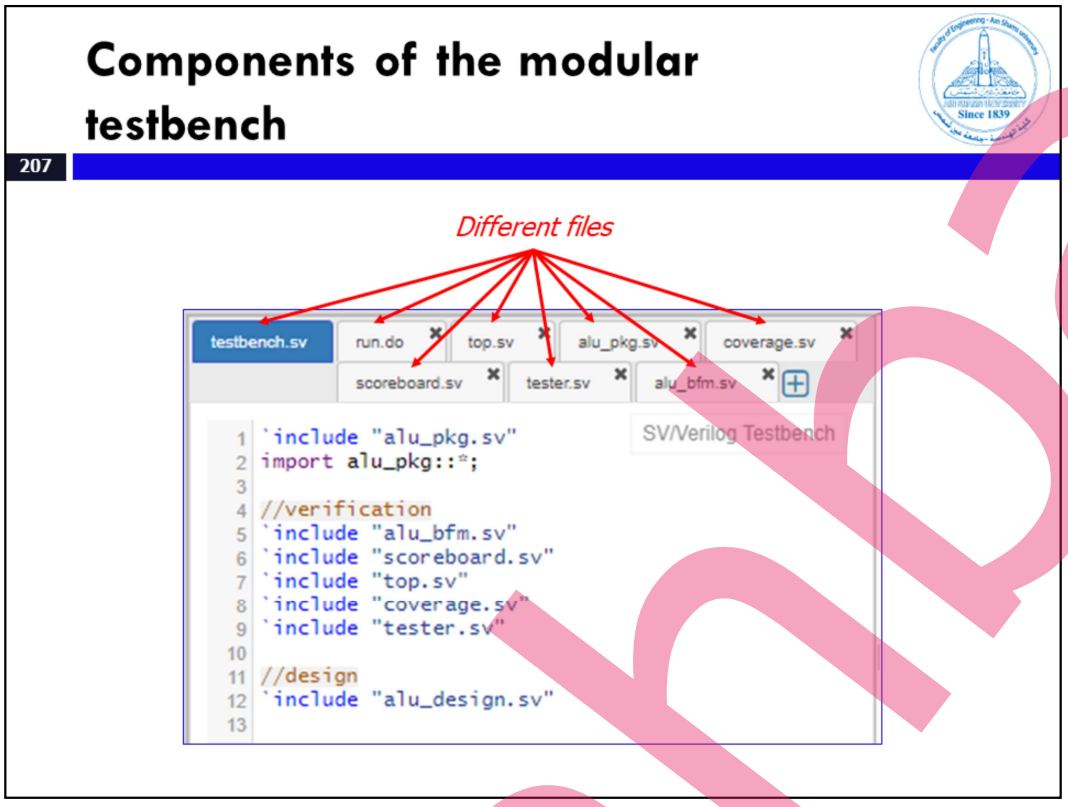
- In the previous part, we created a pretty good testbench for the ALU. It featured all the elements of a modern testbench:
  - Functional Coverage Goals: The testbench measures what we've tested rather than relying upon a list of tests.
  - Self-Checking: We don't have to examine waveforms or look at text output. We can run as many tests as we like.
  - Constrained Random Stimulus: We didn't have to write tests for all the operations; we simply created random stimulus that fulfilled the coverage requirements.
- The downside of our testbench was its *lack of modularity*. This makes it difficult to modify, reuse, or debug that testbench.
- Verification teams have two choices. They can either design testbenches that get more buggy and brittle as the project grows, or they can design testbenches that get more adaptable and powerful as the design grows.
- The testbench we wrote in the previous part was of the first type. That testbench will not grow gracefully with a design.
- By the time our project gets into crunch time, we'll have a testbench that's so brittle that nobody dares touch it because it breaks at the slightest modification.



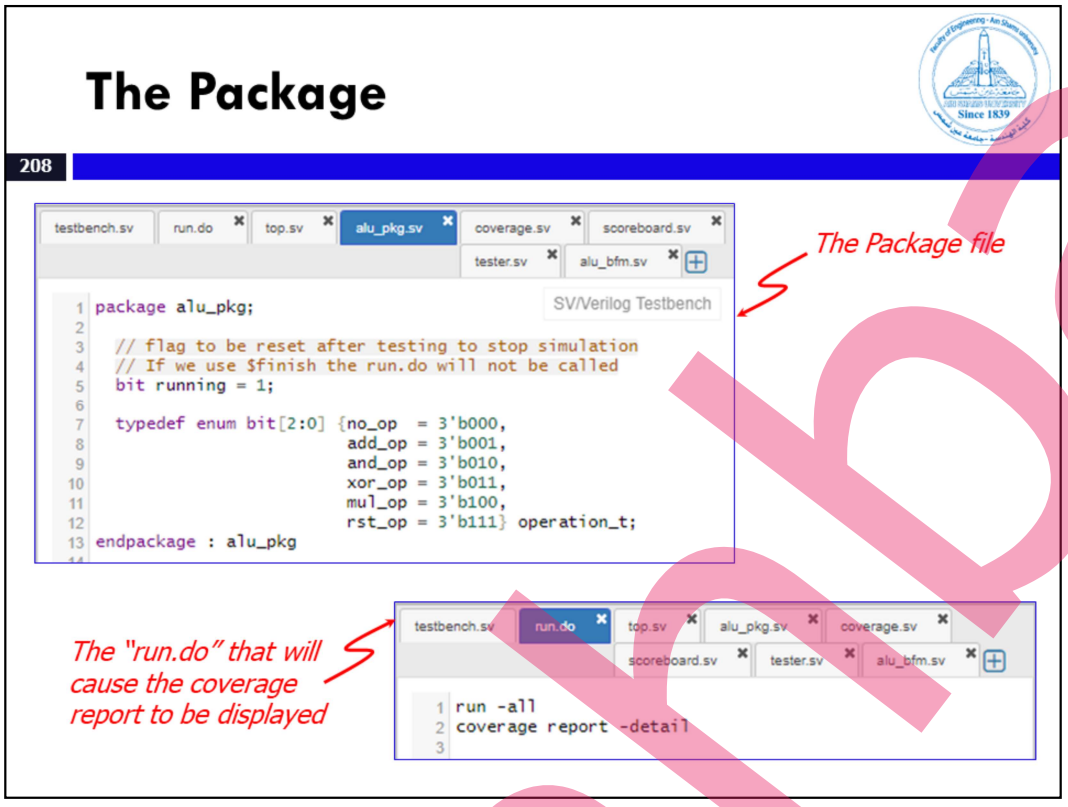
- Engineers need a standardized way to create modular testbenches that grow more powerful over time.
- We want a testbench that grows stronger each time we add a resource and gives future developers easy ways to mix and match features to create new functionality.
- Fortunately, SystemVerilog gives us the tools to do the job.
- The SystemVerilog *interface* is the first of these tools.
- Our first step on the road to the UVM is to modularize our testbench using SystemVerilog interfaces.
- We'll see that interfaces allow us to go beyond simply sharing signals.
- We can also use them to create a Bus Functional Model.



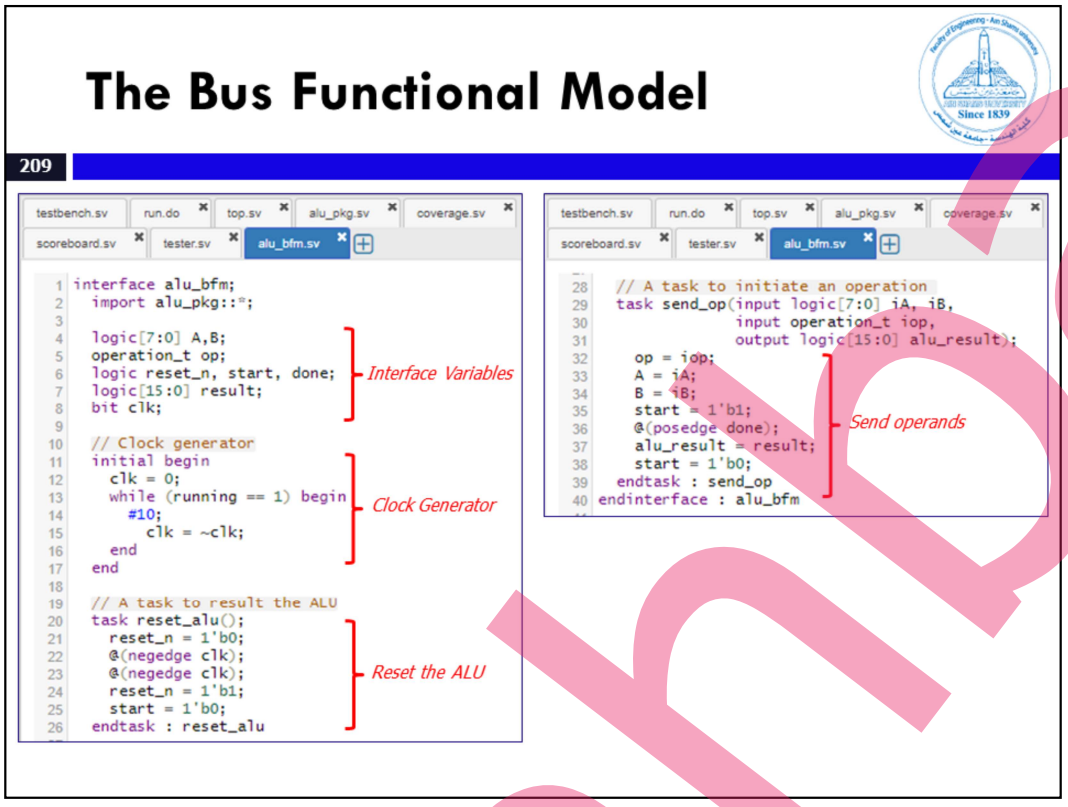
- Engineers need a standardized way to create modular testbenches that grow more powerful over time.
- We want a testbench that grows stronger each time we add a resource and gives future developers easy ways to mix and match features to create new functionality.
- Fortunately, SystemVerilog gives us the tools to do the job.
- The SystemVerilog *interface* is the first of these tools.
- Our first step on the road to the UVM is to modularize our testbench using SystemVerilog interfaces.
- We'll see that interfaces allow us to go beyond simply sharing signals.
- We can also use them to create a Bus Functional Model.



- As we see above, the testbench will simply include all that parts of our test environment using **`include** keyword.
- The first file to be included is the package file, where we define global variables, and types.
- Packages provide a mechanism for storing and sharing data, methods, property, parameters that can be re-used in multiple other modules, interfaces or programs.
- Packages can then be *imported* where items in that package can be used.
- This is done using the keyword **import** followed by the scope resolution operator **::** that then specifies what to import.
- In the shown example, we import everything defined in the package as indicated by the star * that follows **::** operator, to be able to use any of the items.



- In this package file, we define the flag “running” which will cause the clock generation to stop when it is 0, and hence the simulation will stop, and the commands in the file “run.do” will be launched.
- The “run.do” file will cause the coverage report to be generated.



- The *alu_bfm* encapsulates *all the signals* in the ALU testbench and provides a *clock*, a *reset_alu() task*, and a *send_op() task* that sends an operation to the ALU.
- The BFM provides our first step towards modularization. It handles all the signal level stimulus issues so that the rest of our testbench can ignore these issues. For example, the alu_bfm generates the clock. This modularization provides immediate benefits when we break the testbench into modules.
- The type *operation_t* is an enumerated type defined in the package *alu_pkg*. We imported *alu_pkg* at the top of the interface.
- The BFM provides two tasks: *reset_alu()* and *send_op()*.
  - The reset_alu() task drops the reset signal, waits a couple of clocks, and raises it again.
  - The send_op() task sends an operation into the ALU and returns the results. The send_op() task demonstrates how a BFM encapsulates the protocols associated with a DUT. In this case, the task places an operation on the op bus and data on the operand busses. Then it raises the start signal and lowers it based upon the operation requested.
- Encapsulating this behavior has two benefits:
  - We don't need to sprinkle our code with protocol-level behavior; code that calls this task is simpler than code that would have to handle the signals.
  - We can modify all the protocol level behavior in one place; a fix here gets propagated throughout the design.
- Now that we have a ALU BFM, we can modularize the rest of the design.

Tasks versus Functions

210

Function	Task
Execute in one simulation step (No time-consuming statements)	Can have time-consuming statements
Cannot call a task	Can call a function
Can return a value	Cannot return a value
Can be used as an operand in an expression	Cannot be used as an operand in an expression

- All the statements of a function must execute in the same time step, i.e. inside a function you can't have time controlling statements in it (like *wait*, @, …etc). Tasks han have time controlling statements in it.
- Because of the reason above, a functions can't call tasks, while tasks can call other functions and tasks.
- A function can return a single value to the calling point, while a task cannot return a value. Therefore non void functions can be used as an operand in an expression, and the value of the operand will be the value returned by the function.
- In general, functions are extensively used in both design and testbench coding, but tasks are used only in testbench coding as tasks are generally not synthesizable.

The Design module – 1

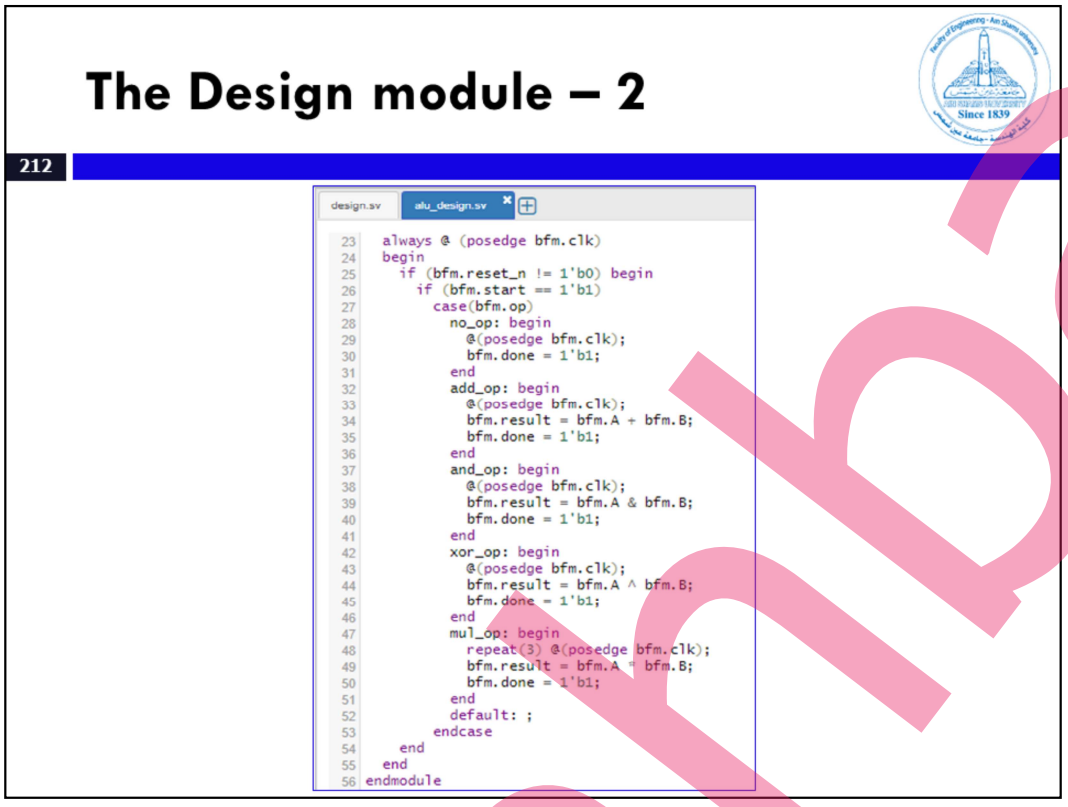
211

design.sv | alu_design.sv

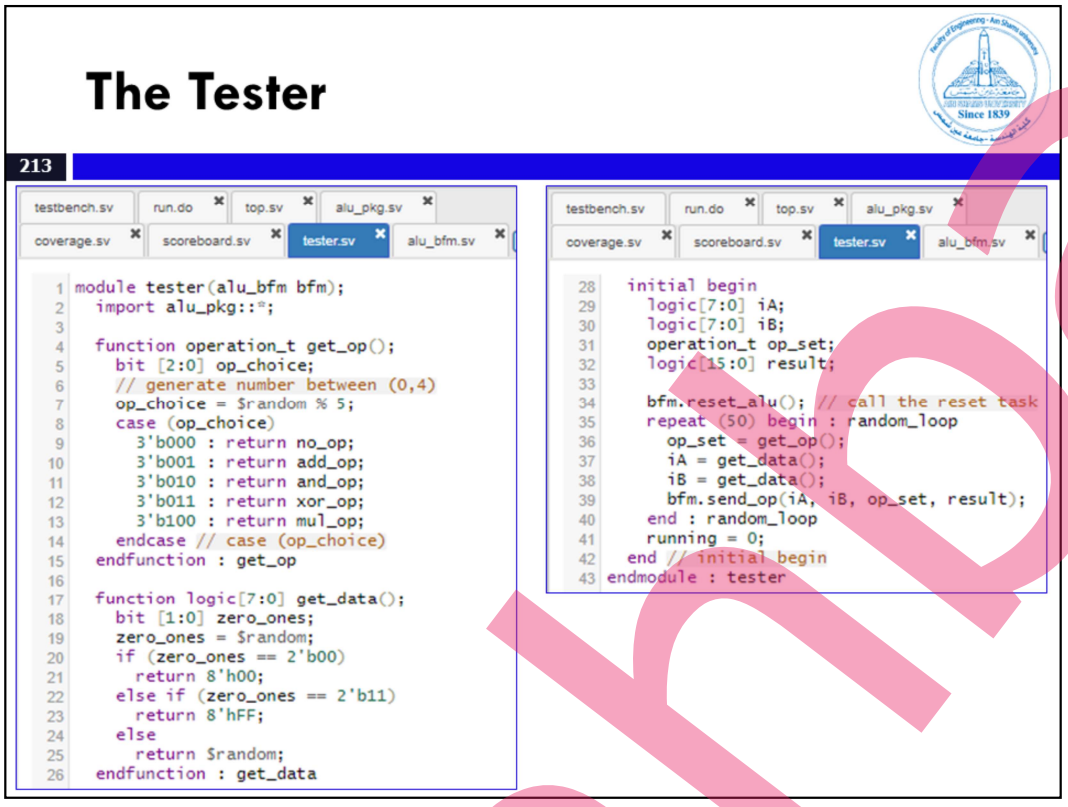
```
1  module alu(alu_bfm bfm);
2
3    initial begin
4      $dumpfile("test.vcd");
5      $dumpvars;
6    end
7
8    always @(posedge bfm.clk)
9    begin
10     if (bfm.reset_n == 0) begin
11       bfm.result = 16'b0000000000000000;
12       bfm.done = 1'b0;
13     end
14   end
15
16   always @ (posedge bfm.clk)
17   begin
18     if (bfm.done == 1) begin
19       bfm.done = 1'b0;
20     end
21   end
```

- Here is another way of generating random values, using $random.
- The first method we have seen before was depending on using the keywork *rand* before declaring a variable, and then calling the function *randomize()* on it.
- Note that $random generates a random integer.
- Here, we want to generate a random operation. As we have only five operations, we used "$random % 5", where % is the mod operation which returns the remainder of the division operation, and thus guarantees the value is between 0 and 4.
- For the function get_data(), we need to get random values for the operands, but we also need to get the corner values "00000000" and "11111111", which are almost impossible to get using pure random number generation.
- We used a simple trick, We generate random values for a 2-bit number. So its possible values are 00, 01, 10, or 11. The probability of getting any one of these values is ¼.
- If the value is 00, the returned value will be "00000000". If it is 11, the returned value will be "11111111", otherwise a random 8-bit value is returned from the function.
- Thus in 25% of the cases, "00000000" is generated, and in 25% of the cases "11111111" is generated, guaranteeing that the corner values are generated easily.
- In the remining 50% of the cases other random operand values are generated.

The Design module – 2



- Here is another way of generating random values, using $random.
- The first method we have seen before was depending on using the keywork *rand* before declaring a variable, and then calling the function *randomize()* on it.
- Note that $random generates a random integer.
- Here, we want to generate a random operation. As we have only five operations, we used "$random % 5", where % is the mod operation which returns the remainder of the division operation, and thus guarantees the value is between 0 and 4.
- For the function get_data(), we need to get random values for the operands, but we also need to get the corner values "00000000" and "11111111", which are almost impossible to get using pure random number generation.
- We used a simple trick, We generate random values for a 2-bit number. So its possible values are 00, 01, 10, or 11. The probability of getting any one of these values is ¼.
- If the value is 00, the returned value will be "00000000". If it is 11, the returned value will be "11111111", otherwise a random 8-bit value is returned from the function.
- Thus in 25% of the cases, "00000000" is generated, and in 25% of the cases "11111111" is generated, guaranteeing that the corner values are generated easily.
- In the remining 50% of the cases other random operand values are generated.



- Here is another way of generating random values, using $random.
- The first method we have seen before was depending on using the keywork *rand* before declaring a variable, and then calling the function *randomize()* on it.
- Note that $random generates a random integer.
- Here, we want to generate a random operation. As we have only five operations, we used "$random % 5", where % is the mod operation which returns the remainder of the division operation, and thus guarantees the value is between 0 and 4.
- For the function get_data(), we need to get random values for the operands, but we also need to get the corner values "00000000" and "11111111", which are almost impossible to get using pure random number generation.
- We used a simple trick, We generate random values for a 2-bit number. So its possible values are 00, 01, 10, or 11. The probability of getting any one of these values is ¼.
- If the value is 00, the returned value will be "00000000". If it is 11, the returned value will be "11111111", otherwise a random 8-bit value is returned from the function.
- Thus in 25% of the cases, "00000000" is generated, and in 25% of the cases "11111111" is generated, guaranteeing that the corner values are generated easily.
- In the remining 50% of the cases other random operand values are generated.

The Scoreboard

214

testbench.sv | run.do | top.sv | alu_pkg.sv | coverage.sv | scoreboard.sv | tester.sv | alu_bfm.sv

SV/Verilog Testbench

```
1  module scoreboard(alu_bfm bfm);
2     import alu_pkg::*;
3
4     always @(posedge bfm.done) begin
5       logic[15:0] predicted_result;
6       case (bfm.op)
7          add_op: predicted_result = bfm.A + bfm.B;
8          and_op: predicted_result = bfm.A & bfm.B;
9          xor_op: predicted_result = bfm.A ^ bfm.B;
10         mul_op: predicted_result = bfm.A * bfm.B;
11      endcase // case (op_set)
12
13      if (bfm.op != no_op)
14        assert(predicted_result == bfm.result)
15          $display("The output is correct");
16        else
17          $error("Wrong %b %s %b = %b",bfm.A, bfm.op, bfm.B,bfm.result);
18    end
19 endmodule : scoreboard
```

- This is the scoreboard, than compares the correct output with the result generated from the design.
- We use an assertion to issue an error message whenever the result is erroneous.
- Normally we don't display a message when the result is correct. We only put it here for illustrating that the assertion may have a pass message and a fail message.

The Coverage module – 1

testbench.sv | run.do | top.sv | alu_pkg.sv | coverage.sv | scoreboard.sv | tester.sv | alu_bfm.sv

```
1  module coverage(alu_bfm bfm);
2    import alu_pkg::*;
3
4    logic[7:0] A, B;
5    operation_t op_set;
6
7    covergroup op_cov;
8      coverpoint op_set {
9        bins single_cycle[] = {[no_op : xor_op]};
10       bins multi_cycle = {mul_op};
11
12       bins sngl_mul[] = ([no_op:xor_op] => mul_op);
13       bins mul_sngl[] = (mul_op => [no_op:xor_op]);
14
15       bins twoops[] = ([no_op:mul_op] [* 2]);
16       bins manymult = (mul_op [* 3:5]);
17       }
18   endgroup
```

- T

The Coverage module – 2

testbench.sv | run.do | top.sv | alu_pkg.sv | coverage.sv | scoreboard.sv | tester.sv | alu_bfm.sv

```
20  covergroup zeros_or_ones_on_ops;
21    all_ops: coverpoint op_set {       // 4 bins
22      ignore_bins null_ops = {no_op};}
23
24    a_leg: coverpoint A {              // 3 nins
25      bins zeros = {'h00};
26      bins others= {['h01:'hFE]};
27      bins ones  = {'hFF};}
28
29    b_leg: coverpoint B {              // 3 bins
30      bins zeros = {'h00};
31      bins others= {['h01:'hFE]};
32      bins ones  = {'hFF};}
33
34    op_00_FF:  cross a_leg, b_leg, all_ops {
35      bins add_00 = binsof (all_ops) intersect {add_op} &&
36                    (binsof (a_leg.zeros) || binsof (b_leg.zeros));
37      bins add_FF = binsof (all_ops) intersect {add_op} &&
38                    (binsof (a_leg.ones) || binsof (b_leg.ones));
39      bins and_00 = binsof (all_ops) intersect {and_op} &&
40                    (binsof (a_leg.zeros) || binsof (b_leg.zeros));
41      bins and_FF = binsof (all_ops) intersect {and_op} &&
42                    (binsof (a_leg.ones) || binsof (b_leg.ones));
43      bins xor_00 = binsof (all_ops) intersect {xor_op} &&
44                    (binsof (a_leg.zeros) || binsof (b_leg.zeros));
45      bins xor_FF = binsof (all_ops) intersect {xor_op} &&
46                    (binsof (a_leg.ones) || binsof (b_leg.ones));
47      bins mul_00 = binsof (all_ops) intersect {mul_op} &&
48                    (binsof (a_leg.zeros) || binsof (b_leg.zeros));
49      bins mul_FF = binsof (all_ops) intersect {mul_op} &&
50                    (binsof (a_leg.ones) || binsof (b_leg.ones));
51      bins mul_max = binsof (all_ops) intersect {mul_op} &&
52                    (binsof (a_leg.ones) && binsof (b_leg.ones));
53      ignore_bins others_only = binsof(a_leg.others) && binsof(b_leg.others);}
54  endgroup
```

- T

The Coverage module – 3

testbench.sv | run.do | top.sv | alu_pkg.sv | coverage.sv | scoreboard.sv | tester.sv | alu_bfm.sv

```
56    op_cov oc;
57    zeros_or_ones_on_ops c_00_FF;
58
59    initial begin : coverage_block
60      oc = new();
61      c_00_FF = new();
62      forever begin  : sampling_block
63        @(negedge bfm.clk);
64        A = bfm.A;
65        B = bfm.B;
66        op_set = bfm.op;
67        oc.sample();
68        c_00_FF.sample();
69      end : sampling_block
70    end : coverage_block
71  endmodule : coverage
```

- T

The Coverage Report – 1

	Metric	Goal	Status
% Hit:	84.21%	100	
bin single_cycle[no_op]	27	1	Covered
bin single_cycle[add_op]	22	1	Covered
bin single_cycle[and_op]	16	1	Covered
bin single_cycle[xor_op]	17	1	Covered
bin multi_cycle	40	1	Covered
bin sngl_mul[xor_op=>mul_op]	0	1	ZERO
bin sngl_mul[and_op=>mul_op]	1	1	Covered
bin sngl_mul[add_op=>mul_op]	2	1	Covered
bin sngl_mul[no_op=>mul_op]	2	1	Covered
bin mul_sngl[mul_op=>xor_op]	0	1	ZERO
bin mul_sngl[mul_op=>and_op]	0	1	ZERO
bin mul_sngl[mul_op=>add_op]	3	1	Covered
bin mul_sngl[mul_op=>no_op]	2	1	Covered
bin twoops[mul_op[*2]]	35	1	Covered
bin twoops[xor_op[*2]]	12	1	Covered
bin twoops[and_op[*2]]	11	1	Covered
bin twoops[add_op[*2]]	15	1	Covered
bin twoops[no_op[*2]]	18	1	Covered
bin manymult	30	1	Covered

- T

The Coverage Report – 2

	Metric	Goal	Status
Coverpoint all_ops	80.00%	100	Uncovered
% Hit:	80.00%	100	
ignore_bin null_ops	27		Occurred
bin auto[add_op]	22	1	Covered
bin auto[and_op]	16	1	Covered
bin auto[xor_op]	17	1	Covered
bin auto[mul_op]	40	1	Covered
bin auto[rst_op]	0	1	ZERO
Coverpoint a_leg	100.00%	100	Covered
% Hit:	100.00%	100	
bin zeros	18	1	Covered
bin others	80	1	Covered
bin ones	23	1	Covered
Coverpoint b_leg	100.00%	100	Covered
% Hit:	100.00%	100	
bin zeros	29	1	Covered
bin others	70	1	Covered
bin ones	22	1	Covered

	Metric	Goal	Status
Cross op_00_FF	47.05%	100	Uncovered
covered/total bins:	8	17	
missing/total bins:	9	17	
% Hit:	47.05%	100	
Auto, Default and User Defined Bins:			
bin add_00	12	1	Covered
bin add_FF	4	1	Covered
bin and_00	4	1	Covered
bin and_FF	2	1	Covered
bin xor_00	7	1	Covered
bin xor_FF	13	1	Covered
bin mul_00	12	1	Covered
bin mul_FF	12	1	Covered
bin mul_max	0	1	ZERO
bin <*,*,*>	0	1	ZERO
Illegal and Ignore Bins:			
ignore_bin others_only	42		Occurred

- T