



2

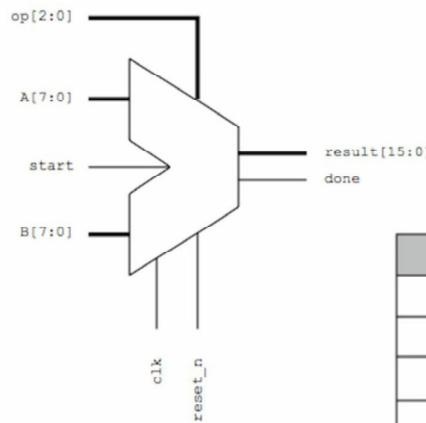
ALU Case Study

- Ver



Simple ALU

3



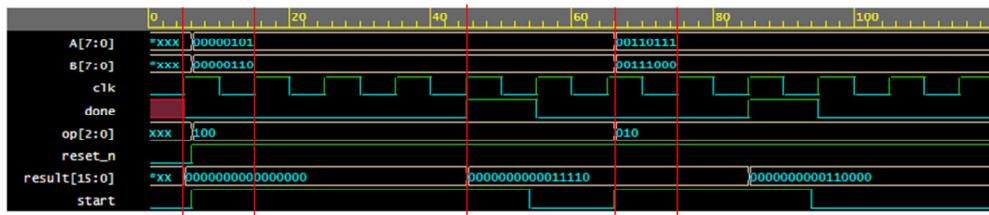
Operation	Opcode
no_op	3'b000
add_op	3'b001
and_op	3'b010
xor_op	3'b011
mul_op	3'b100
unused	3'b101-3'b111

- We will create a complete test environment, by verifying a simple design of an ALU.
- The simple ALU 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 ALU 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 ALU raises the *done* signal when the operation is complete.
- The *reset_n* signal is an active-low, synchronous reset.
- The ALU has five operations: NOP, ADD, AND, XOR, and MULT. The user encodes the operations on the three bit *op* bus when requesting a calculation.



Sample of the ALU Generated Waveforms

4



- The start signal must remain high, and the operator and operands must remain stable until the ALU raises the done signal.
- The done signal stays high for only one clock.
- We'll start our journey by creating a conventional testbench of the ALU. 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)

5

```
1 interface aluifc(input bit clk);
2   logic[7:0] A,B;
3   operation_t op;
4   logic reset_n, start, done;
5   logic[15:0] result;
6   modport tb(output A,B,op,reset_n,start, input done, result,clk);
7   modport dut(input A,B,op,reset_n,start,clk, output done, result);
8 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 issued, $start$
 - 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)

6

```

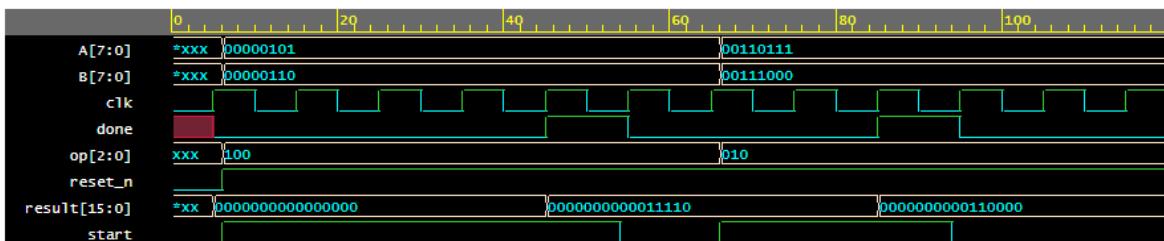
38 module test(aluifc.tb y);
39   initial begin
40     y.reset_n = 0;
41     y.start = 0;
42
43     # 6;
44     y.reset_n = 1;
45     y.A = 8'b000000101;
46     y.B = 8'b000000110;
47     y.start = 1;
48     y.op = mul_op;      // Multiply operation
49
50     #60
51     y.A = 8'b00110111;
52     y.B = 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

```

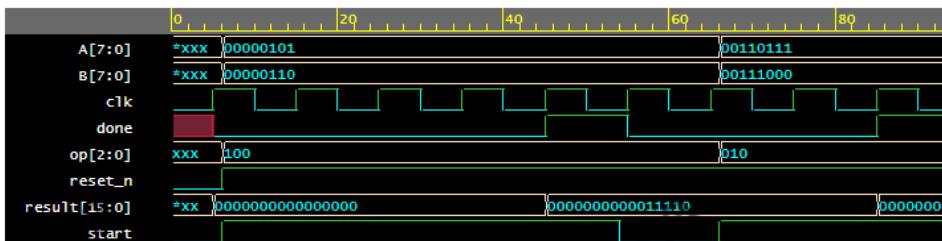
- This is a very simple testbench driving the values of some inputs to the DUT just to visualize some waveforms to make a fast preliminary check.
- Practical note:**

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)



7

```
1 bit running = 1;
2 typedef enum bit[2:0] {
3     no_op = 3'b000,
4     add_op = 3'b001,
5     and_op = 3'b010,
6     xor_op = 3'b011,
7     mul_op = 3'b100} operation_t;
8
9 module alu(aluifc.dut x);
10    always @(posedge x.clk)
11    begin
12        if (x.reset_n == 0) begin
13            x.result = 16'b0000000000000000;
14            x.done = 1'b0;
15        end
16    end
17
18    always @ (posedge x.clk)
19    begin
20        if (x.reset_n != 1'b0) begin
21            if (x.start == 1'b1)
22                case(x.op)
23                    no_op: x.done = 1'b1;
24                    add_op: begin
25                        @(posedge x.clk) x.result = x.A + x.B;
26                        x.done = 1'b1;
27                    end
28                    and_op: begin
29                        @(posedge x.clk) x.result = x.A & x.B;
30                        x.done = 1'b1;
31                    end
32                    xor_op: begin
33                        @(posedge x.clk) x.result = x.A ^ x.B;
34                        x.done = 1'b1;
35                    end
36                    mul_op: begin
37                        repeat(3) @(posedge x.clk);
38                        x.result = x.A * x.B;
39                        x.done = 1'b1;
40                    end
41                default: ;
42            endcase
43        end
44    end
45 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)

8

```
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.



Things to be covered in the test

9

- Test all operations
- Run a multiply after a single cycle operation
- Run a single cycle operation after a multiply
- Simulate all operations twice in a row
- Simulate multiply operation from 3 to 5 times in a row
- Simulate all zeros on an input for all operations
- Simulate all ones on an input for all operations
- Execute all operations after a reset

- 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.

```

1 bit running = 1;
2 typedef enum bit[2:0] {no_op = 3'b000,
3     add_op = 3'b001,
4     and_op = 3'b010,
5     xor_op = 3'b011,
6     mul_op = 3'b100} operation_t;
7
8 //-----
9 interface aluifc(input bit clk);
10    logic[7:0] A,B;
11    operation_t op;
12    logic reset_n, start, done;
13    logic[15:0] result;
14    modport tb(output A,B,op,reset_n,start, input done, result,clk);
15    modport dut(input A,B,op,reset_n,start,clk, output done, result);
16 endinterface
17
18 //-----
19 module top;
20    bit clk;
21    initial begin
22        clk = 0;
23        while(running == 1)
24            #5 clk = ~clk;
25    end
26
27    aluifc u1(clk);
28    alu u2(u1.dut);
29    test u3(u1.tb);
30
31    initial begin
32        $dumpfile("alu.vcd");
33        $dumpvars;
34    end
35 endmodule

```

```

39 module alu(aluifc.dut x);
40    always @ (posedge x.clk)
41    begin
42        if (x.reset_n == 0) begin
43            x.result = 16'b00000000000000000000;
44            x.done = 1'b0;
45        end
46    end
47
48    always @ (posedge x.clk)
49    begin
50        if (x.done == 1) begin
51            x.done = 1'b0;
52        end
53    end
54
55    always @ (posedge x.clk)
56    begin
57        if (x.reset_n != 1'b0) begin
58            if (x.start == 1'b1)
59                case(x.op)
60                    no_op: x.done = 1'b1;
61                    add_op: begin
62                        @(posedge x.clk) x.result = x.A + x.B;
63                        x.done = 1'b1;
64                    end
65                    and_op: begin
66                        @(posedge x.clk) x.result = x.A & x.B;
67                        x.done = 1'b1;
68                    end
69                    xor_op: begin
70                        @(posedge x.clk) x.result = x.A ^ x.B;
71                        x.done = 1'b1;
72                    end
73                    mul_op: begin
74                        repeat(3) @(posedge x.clk);
75                        x.result = x.A * x.B;
76                        x.done = 1'b1;
77                    end
78                    default: ;
79                endcase
80            end
81        end
82    endmodule

```



A first coverage block

11

```
84 module test(aluifc.tb y);
85   class Transaction;
86     rand logic[7:0] operand1, operand2;
87     rand operation_t operation;
88   endclass
89
90   covergroup op_cov @(`posedge y.done);
91   coverpoint y.op {
92     bins single_cycle[] = {[no_op : xor_op]};
93     bins multi_cycle = {mul_op};
94
95     bins sngl_mul[] = ([no_op:xor_op] => mul_op);
96     bins mul_sngl[] = (mul_op => [no_op:xor_op]);
97
98     bins twoops[] = ([no_op:mul_op] [* 2]);
99     bins manymult = (mul_op [* 3:5]);
100   endgroup
101
102   initial begin
103     op_cov xyz;
104     Transaction tr;
105     xyz = new();
106     tr = new();
107
108     y.reset_n = 0;
109     y.start = 0;
110     #7 y.reset_n = 1;
111     y.start = 1;
112
113     repeat (10) begin
114       #2;
115       assert(tr.randomize);
116       y.A = tr.operand1;
117       y.B = tr.operand2;
118       y.op = tr.operation;
119       @(posedge y.done);
120     end
121     #20;
122     running = 0;
123   endmodule
124
```

- 4 bins for single cycle operations
- 1 bin for the multicycle operations

- 5 bins for two repeated operations.
- 1 bins for 3, 4, or 5 repeated multiply operations

- 4 bins for transition from single cycle to multi cycle operations
- 4 bins for transition from multi cycle to single cycle operations

- 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:



A second coverage block

12

```
covergroup zeros_or_ones_on_ops @(posedge y.done);
    a_leg: coverpoint y.A {
        bins zeros = {'h00};
        bins others= {'[h01:'hFE]};
        bins ones  = {'hFF};
    }
    b_leg: coverpoint y.B {
        bins zeros = {'h00};
        bins others= {'[h01:'hFE]};
        bins ones  = {'hFF};
    }
endgroup

initial begin
    op_cov xyz;
    zeros_or_ones_on_ops abc;
    Transaction tr;
    xyz = new();
    abc = new();
    tr = new();

```

- 1 bin to collect the occurrences of 00000000 on operand A
- 1 bin to collect the occurrences of any value between 00000000, and 11111111 on operand A
- 1 bin to collect the occurrences of 11111111 on operand A

• Don't forget to add these lines in your code

- 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

13

```
covergroup zeros_or_ones_on_ops @(posedge y.done);
    all_ops : coverpoint y.op {           // 4 bins
        ignore_bins null_ops = {no_op}; }
    a_leg: coverpoint y.A {
        bins zeros = {'h00};
        bins others= {'h01:'hFE];
        bins ones = {'hFF}; }
    b_leg: coverpoint y.B {
        bins zeros = {"h00"};
        bins others= {"h01:'hFE];
        bins ones = {'hFF}; }
    op_00_FF: cross a_leg, b_leg, all_ops {           //9 bins
        bins add_00 = binsof (all_ops) intersect {add_op} &&
            (binsof (a_leg.zeros) || binsof (b_leg.zeros));
        bins add_FF = binsof (all_ops) intersect {add_op} &&
            (binsof (a_leg.ones) || binsof (b_leg.ones));
        bins and_00 = binsof (all_ops) intersect {and_op} &&
            (binsof (a_leg.zeros) || binsof (b_leg.zeros));
        bins and_FF = binsof (all_ops) intersect {and_op} &&
            (binsof (a_leg.ones) || binsof (b_leg.ones));
        bins xor_00 = binsof (all_ops) intersect {xor_op} &&
            (binsof (a_leg.zeros) || binsof (b_leg.zeros));
        bins xor_FF = binsof (all_ops) intersect {xor_op} &&
            (binsof (a_leg.ones) || binsof (b_leg.ones));
        bins mul_00 = binsof (all_ops) intersect {mul_op} &&
            (binsof (a_leg.zeros) || binsof (b_leg.zeros));
        bins mul_FF = binsof (all_ops) intersect {mul_op} &&
            (binsof (a_leg.ones) || binsof (b_leg.ones));
        bins mul_max = binsof (all_ops) intersect {mul_op} &&
            (binsof (a_leg.ones) && binsof (b_leg.ones));
        ignore_bins others_only = binsof(a_leg.others) && binsof(b_leg.others); }
    endgroup
```

- After 100 runs, only 1 bin of those 9 bins was filled.
- With 500 runs, only 2 bins were filled.
- With 1000 runs, only 5 bins were filled.
- With 2000 runs, 8 bins were filled.
- We could only fill the 9 bins after 936,000 run.
- We had to go from 2,000 to 936,000 just to file the abstinent bin `mul_max`
- This is a typical scenario where we had to go to a directed test to fill the 9th bin.

- 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`: will check cross coverage of things that should happen together. (9 bins out of 36 possible bins from cross interaction $3 \times 3 \times 4$)
 - 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

14

```
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.



Conclusion

15

- The developed testbench delivers functional coverage
- Everything is mixed-up in one module and one file.
- This mixing up makes the testbench difficult to reuse or build upon.
- How do we improve this testbench?
 - The scoreboard, the coverage, and the testing block define completely different pieces of functionality.
 - They shouldn't be in the same file.
- In the next, we'll break our testbench into several modules, and use bus functional models (BFM).

- 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).



Bus Functional Models

- 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 to touch it because it breaks at the slightest modification.

Interfaces & Bus Functional Models

17



- 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.
- The SystemVerilog interface is very useful to do that.
- Our first step on the road to the UVM is to modularize our testbench using SystemVerilog interfaces.
- Interfaces allow us to go beyond simply sharing signals.
- We can also use interfaces 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.



Components of the modular testbench

18

Bus Functional Model Interface

The Tester

The Scoreboard

The Coverage Block

- E

Components of the modular testbench



19

Different files

```
testbench.sv run.do × top.sv × alu_pkg.sv × coverage.sv ×  
scoreboard.sv × tester.sv × alu_bfm.sv × +  
1 'include "alu_pkg.sv"  
2 import alu_pkg::*;

3
4 //verification
5 'include "alu_bfm.sv"
6 'include "scoreboard.sv"
7 'include "top.sv"
8 'include "coverage.sv"
9 'include "tester.sv"
10
11 //design
12 'include "alu_design.sv"
13
```

SV/Verilog Testbench

- 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.

The Package



20

```
1 package alu_pkg;
2
3 // flag to be reset after testing to stop simulation
4 // If we use $finish the run.do will not be called
5 bit running = 1;
6
7 typedef enum bit[2:0] {no_op = 3'b000,
8 add_op = 3'b001,
9 and_op = 3'b010,
10 xor_op = 3'b011,
11 mul_op = 3'b100,
12 rst_op = 3'b111} operation_t;
13
14 endpackage : alu_pkg
```

The Package file

The "run.do" that will cause the coverage report to be displayed

```
1 run -all
2 coverage report -detail
3
```

- 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 Bus Functional Model

21

```

testbench.sv run.do top.sv alu_pkg.sv coverage.sv
scoreboard.sv tester.sv alu_bfm.sv +
```

```

1 interface alu_bfm;
2 import alu_pkg::*;
3
4 logic[7:0] A,B;
5 operation_t op;
6 logic reset_n, start, done;
7 logic[15:0] result;
8 bit clk;
9
10 // Clock generator
11 initial begin
12   clk = 0;
13   while (running == 1) begin
14     #10;
15     clk = ~clk;
16   end
17 end
18
19 // A task to result the ALU
20 task reset_alu();
21   reset_n = 1'b0;
22   @(negedge clk);
23   @(negedge clk);
24   reset_n = 1'b1;
25   start = 1'b0;
26   endtask : reset_alu

```

Interface Variables

Clock Generator

Reset the ALU

```

testbench.sv run.do top.sv alu_pkg.sv coverage.sv
scoreboard.sv tester.sv alu_bfm.sv +
```

```

28 // A task to initiate an operation
29 task send_op(input logic[7:0] iA,
30               input operation_t iop,
31               output logic[15:0] alu_result);
32   op = iop;
33   A = iA;
34   B = iB;
35   start = 1'b1;
36   @(posedge done);
37   alu_result = result;
38   start = 1'b0;
39   endtask : send_op
40 endinterface : alu_bfm

```

Send operands

- 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.



The Design module – 1

22

```
design.sv alu_design.sv x +  
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
```



The Design module – 2

23

```
design.sv alu_design.sv x +  
23  always @ (posedge bfm.clk)  
24    begin  
25      if (bfm.reset_n != 1'b0) begin  
26        if (bfm.start == 1'b1)  
27          case(bfm.op)  
28            no_op: begin  
29              @(posedge bfm.clk);  
30              bfm.done = 1'b1;  
31            end  
32            add_op: begin  
33              @(posedge bfm.clk);  
34              bfm.result = bfm.A + bfm.B;  
35              bfm.done = 1'b1;  
36            end  
37            and_op: begin  
38              @(posedge bfm.clk);  
39              bfm.result = bfm.A & bfm.B;  
40              bfm.done = 1'b1;  
41            end  
42            xor_op: begin  
43              @(posedge bfm.clk);  
44              bfm.result = bfm.A ^ bfm.B;  
45              bfm.done = 1'b1;  
46            end  
47            mul_op: begin  
48              repeat(3) @(posedge bfm.clk);  
49              bfm.result = bfm.A * bfm.B;  
50              bfm.done = 1'b1;  
51            end  
52            default: ;  
53          endcase  
54        end  
55      endmodule
```



The Tester

24

```
testbench.sv run.do top.sv alu_pkg.sv
coverage.sv scoreboard.sv tester.sv alu_bfm.sv
28 initial begin
29 logic[7:0] iA;
30 logic[7:0] iB;
31 operation_t op_set;
32 logic[15:0] result;
33
34 bfm.reset_alu(); // call the reset task
35 repeat (50) begin : random_loop
36   op_set = get_op();
37   iA = get_data();
38   iB = get_data();
39   bfm.send_op(iA, iB, op_set, result);
40 end : random_loop
41 running = 0;
42 end // initial begin
43 endmodule : tester
1 module tester(alu_bfm bfm);
2 import alu_pkg::*;
3
4 function operation_t get_op();
5   bit [2:0] op_choice;
6   // generate number between (0,4)
7   op_choice = $random % 5;
8   case (op_choice)
9     3'b000 : return no_op;
10    3'b001 : return add_op;
11    3'b010 : return and_op;
12    3'b011 : return xor_op;
13    3'b100 : return mul_op;
14  endcase // case (op_choice)
15 endfunction : get_op
16
17 function logic[7:0] get_data();
18   bit [1:0] zero_ones;
19   zero_ones = $random;
20   if (zero_ones == 2'b00)
21     return 8'h00;
22   else if (zero_ones == 2'b11)
23     return 8'hFF;
24   else
25     return $random;
26 endfunction : get_data
```

- Here is another way of generating random values, using `$random`.
- The first method we have seen before was depending on using the keyword `rand` before declaring a variable, and then calling the function `randomize()` on it.
- Note that `$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 $\frac{1}{4}$.
- 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 remaining 50% of the cases other random operand values are generated.



The Scoreboard

25

```
testbench.sv run.do top.sv alu_pkg.sv coverage.sv scoreboard.sv tester.sv alu_bfm.sv
+
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
```

SV/Verilog Testbench

- This is the scoreboard, that 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

26

```
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 twooops[] = ([no_op:mul_op] [* 2]);
16      bins manymult = (mul_op [* 3:5]);
17    }
18  endgroup
```

- T



The Coverage module – 2

27

```
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 {  
22         ignore_bins null_ops = {no_op}; }  
23  
24     a_leg: coverpoint A {  
25         bins zeros = {'h00};  
26         bins others= {'[h01:hFE]};  
27         bins ones = {'hFF}; }  
28  
29     b_leg: coverpoint B {  
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

28

```
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

29

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

30

	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