The Importance of Path Coverage in Completely Defining Functional Coverage

White Paper

May 2010

SOLID OAK TECHNOLOGIES

www.solidoaktech.com

1. INTRODUCTION

What determines when an ASIC is ready for tape out or an FPGA is ready for production? The basis for this decision has evolved as new methodologies and tools have evolved. Code coverage was one of the early metrics used to determine when testing was complete. While easily integrated into the flow, code coverage quickly proved to be an inadequate metric.

Functional coverage has become the frontrunner for determining verification complete. The arduous task of defining functional coverage presents the next obstacle. In modern flows, engineers have to translate the high level specification into a verification plan, execute it, and hope there's enough coverage (and time) in the plan to ensure a working device.

Solid Oak Technologies has developed a methodology using tools designed to automate the creation of the functional coverage metrics and shorten the time to verification complete. Today a designer can completely define a function with flow diagrams and timing diagrams utilizing Solid Oak's easy to enter and modify graphics tool, CoverAll™, compile them in seconds, and have a complete set of assertions and path covers.

"In general, an assertion is a statement about a design's intended behavior (that is, a property), which must be verified."[1] The assertion is the functional "checker" in verification. It ensures that the correct state was achieved for the assigned function.

The value of the path covers is not as apparent. A path cover is a cover statement which indicates the assigned function was tested. The difference between an assertion and a path cover is subtle. The path cover is a subset of the assertion. An assertion, by definition, must contain all the paths and a functional checker. This means that an assertion which contains multiple path covers can "pass" without testing every path.

Flowcharts can be utilized to completely define all possible paths and state combinations a function can have. CoverAll™ takes this graphical representation and converts it into assertions, sequences and path coverage properties in either the PSL[2] or SVA[3] language, or into an assertion library, such as OVL[4], which can be included in simulations or formal verification. Once all of these cover items have been exercised, the design has 100% functional coverage (i.e. design intent is tested and checked).

This paper details how to create a useful flow diagram to identify the coverage items necessary for full functional coverage with the CoverAll™ toolset. In addition, the method will be illustrated by way of very simple, real design example.

2. Converting a Specification to a Flow Diagram

Once the functional specification is available, the design and verification engineers begin dissecting its sections into modules. As an example, consider the simple register map specification shown below. The table defines 5 registers, their individual addresses, their mode of operation and value at reset.

Address	Mode	Name	Reset Value
3'b000	R/W	reg0	8'h00
3'b001	RO	reg1	8'h00
3'b010	R/W	reg2	8'h00
3'b011	R/W	reg3	8'h00
3'b100	R/W	reg4	8'h00

Table 1: Address Map

In addition to the data presented in the table, the specification also indicates that writing to reg0 causes a signal, 'start', to be asserted and that reading from reg0 causes 'start' to be deasserted. The designer creates the flow diagram, shown in Figure 1, from this specification with the CoverAll™ graphical entry tool.

This flow diagram defines every possible path and state for the address decoder function described in the specification. CoverAll™ logic drawings must also contain a title block (a logic drawing may require multiple pages but needs to contain only one title block) which provides the appropriate switches for compiling the drawing. Refer to the CoverAll™ User Manual for details on all of the allowable switches. For this design, we are targeting the SVA language and placing the generated coverage items in the file 'addr_sv.sv'.

As seen in Figure 1, the address decoder contains an active low synchronous reset which connects the Terminator block (oval) where the clock is defined, to the Process block (rectangle) which contains the reset values. If this design required an asynchronous reset, it would have been defined in the Terminator block as well.

Each Decision block (diamond) has only one 'NO' output and one 'Yes' output. If one of these outputs goes to multiple blocks, an Extender (inverse mux) must be used so that CoverAll™ can automatically detect mutually exclusive paths. In Figure 1, the blocks with the number 2 in them are Extenders. They are used because all of the paths through the Decision blocks with the signal 'addr' in the condition are mutually exclusive. Each Decision block and Process block contain only one input. If more than one path in a function enters the same Decision block or Process block, an Aggregator block (mux) needs to be used. In Figure 1, the blocks with the numbers 1 and 3 on them are Aggregators. Lastly, the end of every path must be connected back to the reset Process block.

It is important to note that every register is assigned a state in every Process block. This ensures that every register has an assertion associated with every path. For example, if only the registers which are updated by a given path were included in the Process block, no assertions for that path would be generated for the other registers. If one of those registers does get updated when that path is tested, it will not be checked.

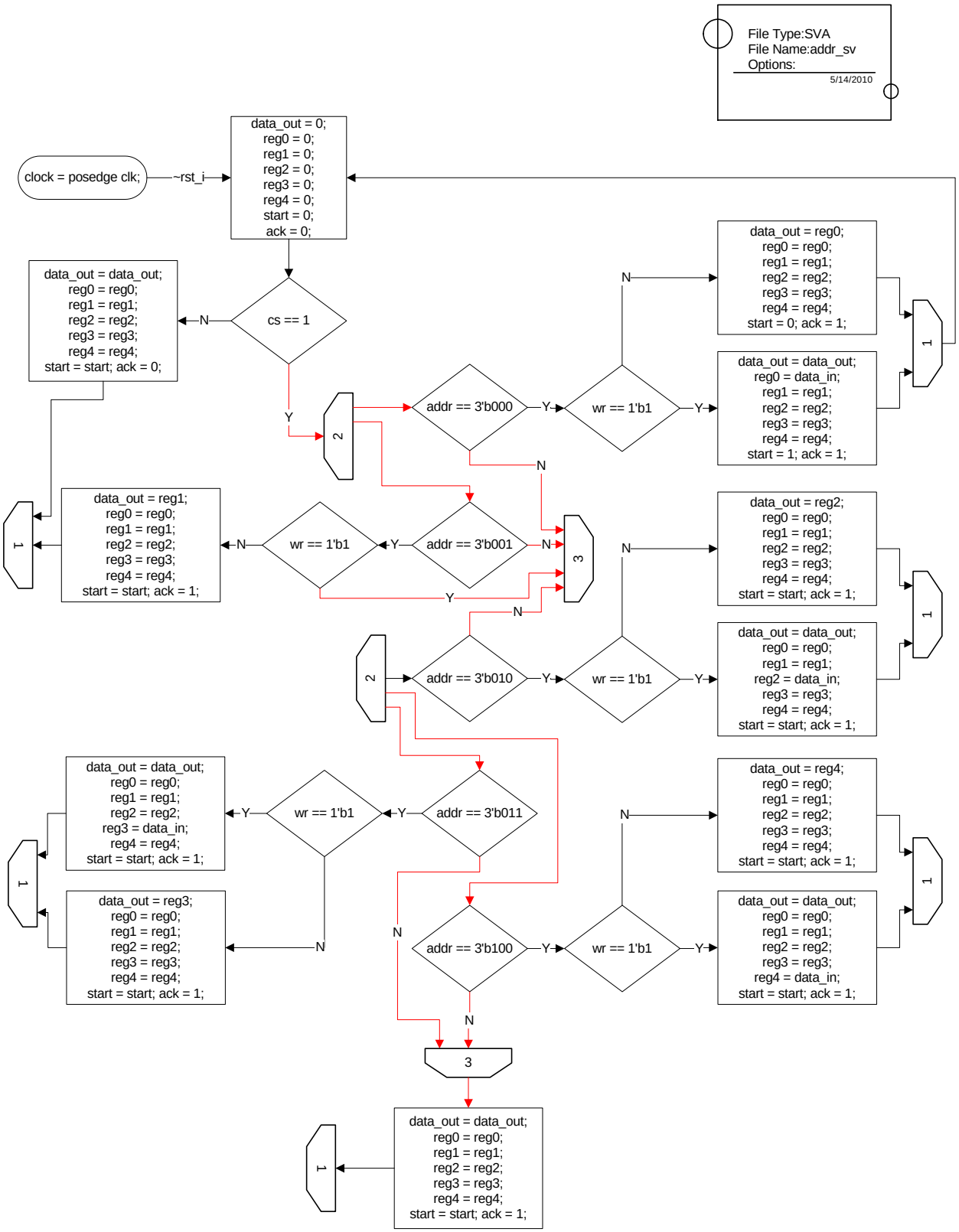

Figure 1: Flow Diagram for the Address Decoder

3. Converting a Flow Diagram into Assertions and Path Covers

With the completed diagram, the designer invokes the CoverAll™ compiler to convert the graphical representation into assertions and path covers. Compiling the example flow diagram of the address decoder above produces the following results:

```
********** Document Statistics **********
*                                       *
************ Flow Statistics *************
* Total Async Resets:          0        *
*  Total Sync Resets:          8        *
*   Total Assertions:         88        *
*  Total Path Covers:         16        *
************* FSM Statistics *************
* Total Async Resets:          0        *
*  Total Sync Resets:          0        *
*  Total Transitions:          0        *
*    Total Sequences:          0        *
*    Total FSM Loops:          0        *
*   Total Hold Terms:          0        *
******* Timing Diagram Statistics ********
*    Total Sequences:          0        *
*      Total Asserts:          0        *
*****************************************
```

Figure 2 – CoverAll™ Statistics

Referring to Figure 1 and Figure 2, the 8 synchronous reset assertions are generated from the 8 registers defined in the reset Process block, one for each signal regardless of its register width. The 88 assertions are generated from the 8 registers in the remaining 11 Process blocks (one per signal, per process block). The 16 path covers are generated from the 10 single paths to 10 of the Process blocks plus the 6 paths, shown in red, which use the number 3 Aggregator to enter the Process block at the bottom of the diagram. The assertions and path covers produced by the CoverAll™ compiler are written to the file specified in the title block. This file is utilized in simulation regressions or as formal verification targets.

4. The Importance of the Path Covers

As shown by the red arrows in Figure 1, there are 6 possible functional paths to the same 8 assertions. The assertion and the 6 path covers for reg0 are shown below.

```
property fc2a_reg0_prop2;
@(posedge clk)
disable iff((~rst_i))
(((wr == 1'b1) & (addr == 3'b001) & (cs == 1)) | ((~(addr == 3'b000) & (cs == 1)) & (~(addr == 3'b001) & (cs == 1)) & (~(addr == 3'b010) & (cs == 1)) & (~(addr == 3'b011) & (cs == 1)) & (~(addr == 3'b100) & (cs == 1))) |=> reg0 === $past(reg0));
endproperty
assert property(fc2a_reg0_prop2);

FC2A_PATH_ADDR_DEC_SV_0_2 : cover property (@(posedge clk) ((wr == 1'b1) & (addr == 3'b001) & (cs == 1)));
FC2A_PATH_ADDR_DEC_SV_0_6 : cover property (@(posedge clk) (~(addr == 3'b000) & (cs == 1)));
FC2A_PATH_ADDR_DEC_SV_0_7 : cover property (@(posedge clk) (~(addr == 3'b001) & (cs == 1)));
FC2A_PATH_ADDR_DEC_SV_0_8 : cover property (@(posedge clk) (~(addr == 3'b010) & (cs == 1)));
FC2A_PATH_ADDR_DEC_SV_0_9 : cover property (@(posedge clk) (~(addr == 3'b011) & (cs == 1)));
FC2A_PATH_ADDR_DEC_SV_0_10 : cover property (@(posedge clk) (~(addr == 3'b100) & (cs == 1)));
```

Figure 3: Assertion and Path Covers for 'reg0'

The fact that multiple path covers are included in one assertion means that an assertion can "pass" without every functional path being tested. If it isn't tested, it will be broken! This can be shown by creating a testbench and running a simulation with bugs strategically placed in the RTL.

4.1 "Buggy" Address Decoder RTL

The following is an example of one possible implementation of the RTL written to represent the flow diagram of the address decoder. Two bugs have been injected by the designer as a result of simple oversights: reg1 is writeable and the 'start' signal is asserted and cleared by both 'addr' equal to 3'b000 and 3'b100 (address aliasing).

```
`include "../timescale.v"
module addr_dec (clk, rst_i, cs, wr, addr, data_in, data_out, start, ack);
parameter awidth = 3;
parameter dwidth = 8;
  input        clk;
  input        rst_i;
  input        cs;
  input        wr;
  input  [awidth-1:0] addr;
  input  [dwidth-1:0] data_in;
  output [dwidth-1:0] data_out;
  output       start;
  output       ack;

  reg [dwidth-1:0] data_out;
  reg         start, ack;
  reg [dwidth-1:0] reg0, reg1, reg2, reg3, reg4;

  always @(posedge clk) begin
    if(rst_i == 1'b0) begin
      data_out <= 0;
      reg0 <= 0;
      reg1 <= 0;
      reg2 <= 0;
      reg3 <= 0;
      reg4 <= 0;
      start <= 0;
      ack <= 0;
    end
    else begin
      case(addr) //synopsys parallel_case
      3'b000: data_out <= (cs && ~wr) ? reg0 : data_out;
      3'b001: data_out <= (cs && ~wr) ? reg1 : data_out;
      3'b010: data_out <= (cs && ~wr) ? reg2 : data_out;
      3'b011: data_out <= (cs && ~wr) ? reg3 : data_out;
      3'b100: data_out <= (cs && ~wr) ? reg4 : data_out;
      default: data_out <= data_out;
      endcase
      reg0 <= (cs && addr == 3'b000 && wr) ? data_in : reg0;
```

```
            // REG1 should be read only
      reg1 <= (cs && addr == 3'b001 && wr) ? data_in : reg1;
      reg2 <= (cs && addr == 3'b010 && wr) ? data_in : reg2;
      reg3 <= (cs && addr == 3'b011 && wr) ? data_in : reg3;
      reg4 <= (cs && addr == 3'b100 && wr) ? data_in : reg4;
// Start should use all 3 bits of addr; otherwise 000 and 100
alias
      start <= (cs && addr[1:0] == 2'b00) ? wr : start;
      ack <= cs;
    end
  end
endmodule
```

Figure 4: RTL Implementation of the Address Decoder

4.2 Simulation of the "Buggy" Address Decoder

A simple testbench with read and write tasks can be used to test the RTL. The simulation will execute in three phases. The first phase will write and read addresses 3'b000 and 3'b010 to show that full code coverage of the 'start' signal can be achieved while a bug in that signal remains. The second phase of the simulation will write and read addresses 3'b011, 3'b100, and 3'b101 (an invalid address) and read address 3'b001. This phase will show the assertions for the 'start' signal will fail when the write and read to address 3'b100 occur. The other more interesting result is that all other assertions pass. This means that if the 'start' signal in the RTL were fixed, all the assertions would indicate that the RTL has all been checked. However, an additional bug is still hidden. The last phase of the simulation will write to address 3'b001. This write results in an assertion error on 'reg1'. At this point all path covers have been reached, all assertions have been checked, both bugs have been found, and complete functional coverage has been achieved.

The testbench used in the simulation is included below.

```
`include "../timescale.v"

module tst_bench_top();
parameter awidth = 3;
parameter dwidth = 8;
reg  clk, rst_i, cs, wr;
reg  [dwidth -1:0] data_in;
reg  [awidth -1:0] addr;
integer i;
wire [dwidth -1:0] data_out;
wire ack, start;
wire [dwidth -1:0] dout;

`ifdef NO_BIND
          addr_dec #(awidth, dwidth) u0 (
                    .clk(clk),
                    .rst_i(rst_i),
                    .cs(cs),
                    .wr(wr),
                    .addr(addr),
                    .data_in(data_in),
                    .data_out(data_out),
                    .start(start),
                    .ack(ack)
          );
`else
          addr_dec #(awidth, dwidth) u0 (.*);
          bind addr_dec addr_dec_mod u1 (.*);
`endif
          always
            #5 clk = ~clk;
          initial
            begin
            $display("\nstatus: %t Testbench started\n\n", $time);
              clk = 0;
              addr = 3'b000;
              cs = 1'b0;
              wr = 1'b0;
              data_in = 8'd0;
                  // reset system
                  rst_i = 1'b1; // negate reset
                  #2 rst_i = 1'b0; // assert reset
                  repeat(2) @(posedge clk);
                  rst_i = 1'b1; // negate reset
                  $display("status: %t done sync reset", $time);
                  repeat(60) @(posedge clk);
                  @(posedge clk);
                  #2 addr = 3'b000;
                  i = 1;
                  for (i = 1; i < 5; i=i+1) begin
                      repeat(2) @(posedge clk);
                      #2; addr = i;
                  end
                  repeat(10) @(posedge clk);
                  reg_write(1, 3'b000, 8'ha5);
                  reg_read(1, 3'b000);
                  reg_write(1, 3'b010, 8'h0f);
                  reg_read(1, 3'b010);
        `ifdef STARTERROR
                  reg_read(1, 3'b001);
                  reg_write(1, 3'b011, 8'hf0);
                  reg_read(1, 3'b011);
                  reg_write(1, 3'b100, 8'hf0);
                  reg_read(1, 3'b100);
                  reg_write(1, 3'b101, 8'hf0);
                  reg_read(1, 3'b101);
        `endif
        `ifdef REG1ERROR
                  reg_write(1, 3'b001, 8'h5a);
        `endif
                  repeat(10) @(posedge clk);
                  $stop;
              end

task reg_read;
    input delay;
    integer delay;
    input [awidth-1:0] a;
    begin
        repeat(delay) @(posedge clk);
        #1 addr = a;
        cs = 1'b1;
        wr = 1'b0;
        @(posedge clk);
        while(~ack) @(posedge clk);
        #1 addr = {awidth{1'bx}};
        cs = 1'b0;
        wr = 1'b0;
    end
endtask
```

```
task reg_write;
    input delay;
    integer delay;
    input [awidth-1:0] a;
    input [dwidth-1:0] d;
    begin
        repeat(delay) @(posedge clk);
        #1 addr = a;
        cs = 1'b1;
        wr = 1'b1;
        data_in = d;
        @(posedge clk);
        while(~ack) @(posedge clk);
        #1 addr = {awidth{1'bx}};
        cs = 1'b0;
        wr = 1'b0;
    end
endtask
endmodule
```

Figure 5: Testbench for Address Decoder

The first phase of the simulation produces 100% code coverage for the 'start' signal by only accessing two registers; however, a bug in this signal still exists.

start <= (cs == 1'b1 && addr[1:0] == 2'b00) ? wr == 1...			100.00%
Branch	**Source**	**Hits**	**Status**
IF	start <= (cs == 1'b1 && addr[1:0] == 2'b00) ? wr == 1...	4	Covered
ELSE	start <= (cs == 1'b1 && addr[1:0] == 2'b00) ? wr == 1...	14	Covered

Figure 6: Code Coverage Results for 'start' Signal

The second part of the simulation results in two assertion failures for the 'start' signal when address 3'b100 aliases to address 3'b000. Otherwise, all assertions would have been attempted and "passed". If this were used to determine that testing was complete, there would still be one more bug. Notice there is still one directive, a path cover, not yet covered.

Weighted Average:				96.39%
Coverage Type	**Bins**	**Hits**	**Misses**	**Coverage (%)**
Directive	16	15	1	93.75%
Statement	22	22	0	100.00%
Branch	30	29	1	96.66%
UDP Condition	38	36	2	94.73%
FEC Condition	54	50	4	92.59%
Assertion Attempted	96	96	0	100.00%
Assertion Passes	96	96	0	100.00%
Assertion Failures	96	2	-	2.08%
Assertion Successes	96	94	2	97.91%

Figure 7: Full Assertion Coverage, Incomplete Path Coverage

Lastly, by running the final phase of the simulation and achieving the last path cover, 100% functional coverage is achieved and both implementation bugs were found.

Weighted Average:				99.37%
Coverage Type	**Bins**	**Hits**	**Misses**	**Coverage (%)**
Directive	16	16	0	100.00%
Statement	22	22	0	100.00%
Branch	30	30	0	100.00%
UDP Condition	38	38	0	100.00%
FEC Condition	54	54	0	100.00%
Assertion Attempted	96	96	0	100.00%
Assertion Passes	96	96	0	100.00%
Assertion Failures	96	3	-	3.12%
Assertion Successes	96	93	3	96.87%

Figure 8: 100% Functional Coverage

5. Conclusion

The conclusion can be drawn from this very simple example that path covers are a necessary functional coverage metric. Once this point is agreed upon, it only stands to reason that the definition of the path covers and assertions becomes the problem to solve. Obviously, as the function complexity increases the difficulty of defining all of the path covers and assertions does too. The creation of flow diagrams and timing diagrams is an easy way to ensure all design intent is defined and documented, regardless of how they are converted into the verification aids. With Solid Oak's methodology and CoverAll™ toolset, no matter how complex, if you can define it to design it, let the debug begin!

6. REFERENCES

[1] Assertion-Based Design 2nd Edition, Foster et al

[2] SystemVerilog Assertions (SVA), Accellera/IEEE, www.systemverilog.org

[3] Property Specification Language (PSL), Accellera/IEEE, www.eda-stds.org/ieee-1850/www.eda.org/vfv

[4] Open Verification Library www.accellera.org/activities/ovl