2ECDE59
Testing and Verification of Digital Circuits

SPECIAL ASSIGNMENT

"ASSERTION BASED VERIFICATION USING JASPERGOLD"

Submitted By:
Aditya Jain (21BEC003)
Shabbir Aglodiya (21BEC006)
Sujal Bhojani (21BEC016)

Introduction to Assertion Based Verification:

An assertion is a statement about your design that you expect to be true always. Assertions are primarily used to validate the behaviour of a design and accordingly warning or errors will be generated on a failure of a condition.

Types of Assertion Statements:

The assertion statement can be of the following types:

- Assert - To specify that the given property of the design is true in simulation.
- Assume - To specify that the given property is an assumption and used by formal tools to generate input stimulus.
- Cover – To evaluate the property for functional coverage.
- Restrict – To specify the property as a constraint on formal verification computations and is ignored by simulators.

Building Block of SVA:

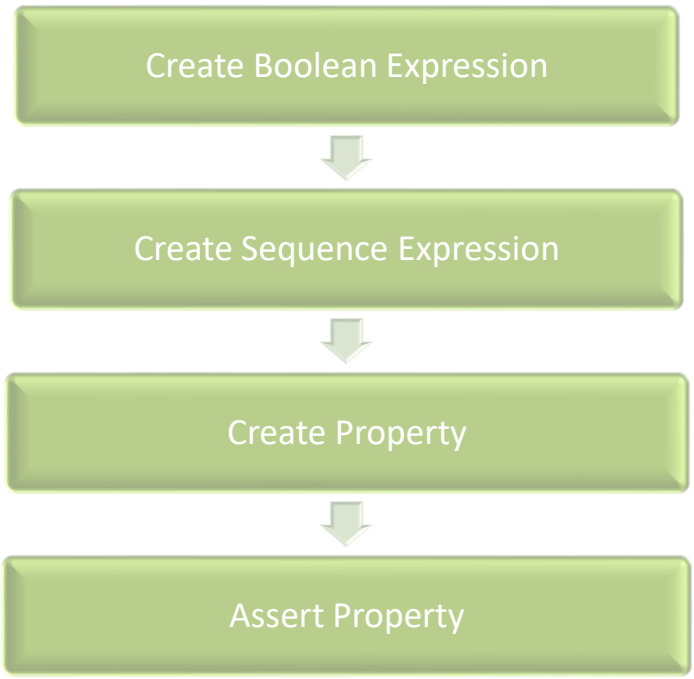

Figure 1 Block Diagram of SVA

Boolean expressions:

The functionality is represented by the combination of multiple logical events. These events could be simple Boolean expressions

Sequence:

A sequence of multiple logical events typically forms the functionality of any design. These events may span across multiple clock or exist for just a single clock cycle.

- **Syntax:**

```
sequence <name_of_sequence>
     <test_expression>
endsequence

assert property (<name_of_sequence>)
```

Property:

Multiple sequences can be logically or sequentially combined to generate more intricate sequences. SVA offers a keyword to denote these complex sequential behaviours, known as "property".

- **Syntax:**

```
property <name_of_property>
     <test_expression> or <sequence expression>
endproperty

assert property (<name_of_property>)
```

Assert:

The property is the one that is verified during a simulation. It has to be asserted to take effect during a simulation. SVA provides a keyword called “assert” to check the property.

➢ **Syntax:**

```
assert name: assert_property(property_name)
```

There are two types of assertions:

- Immediate Assertions
- Concurrent Assertions

Immediate Assertions:

Immediate Assertions are as simple as **if** statement. These assertions do not depend upon a clock edge or reset.

➢ **Syntax:**

```
assert (expression) $display("True Part");
       else  $display("False Part");
```

```
// This is just dummy code. It's an alternate way to think of immediate assertion.
always_comb begin
    if (assertion statement)
        // do nothing;
    else
        $error("fail");
end
```

Figure 2 Immediate Assertion

The above code structure is equivalent to an if- condition within an always_comb block. The assertion checks to make sure current_state is never 0. As the simulation test runs, the simulator continuously checks the expression within assert(). If it returns false, the error message is printed. I usually skip the error message because the simulator prints a default message with the assertion name in the absence of a custom error message.

Concurrent Assertions:

Concurrent assertions let you describe more complex expressions that span time and are triggered relative to a clock edge.

The keyword property distinguishes an immediate from a concurrent assertion. There are two formats:

```
// Format 1 - Inline expression
concurrent_assertion_name:                          // assertion label
    assert
        property (
            @(posedge clk) disable iff (rst)        // sampling event
            req |-> ##3 gnt                         // expression to check
        )
        else                                        // (optional) error message
            $error("%m no grant after request");

// Format 2 - Separate property block
property ConcurrentPropName;
   @(posedge clk) disable iff (rst)
   req |-> ##3 gnt;
endproperty
AssertionName: assert property (ConcurrentPropName);
```

Figure 3 Concurrent Assertion (I)

The above example shows the general structure of a concurrent expression.

- The assertion is disabled during reset
- The variables in the expression are sampled at the edge of the clock specified
- The expression checks if a grant gnt arrives 3 clocks after request req is made.
- If the expression returns false, an error is reported

```
// This is just dummy code. It's an alternate way to think of concurrent assertion.
always_ff @(posedge clk) begin
    if (rst) begin
        // in rst
    end else begin
        if (assertion expression)
            // do nothing;
        else
            $error("fail");
        end
    end
end
```

Figure 4 Concurrent Assertion as if statement within always_ff block

Implication Operator

When you look at examples of concurrent assertions, you'll see |-> and |=> operators frequently used. These are called implication operators and the LHS of the operator is called the antecedent and the RHS is called the consequent.

```
|-> : Overlapping implication - The RHS is evaluated from the same cycle LHS is true
|=> : Non-overlapping implication - The RHS is evaluated one clock cycle after LHS is true
```

Consider the following example waveform. The code below shows how to express this waveform with |-> and |=>.

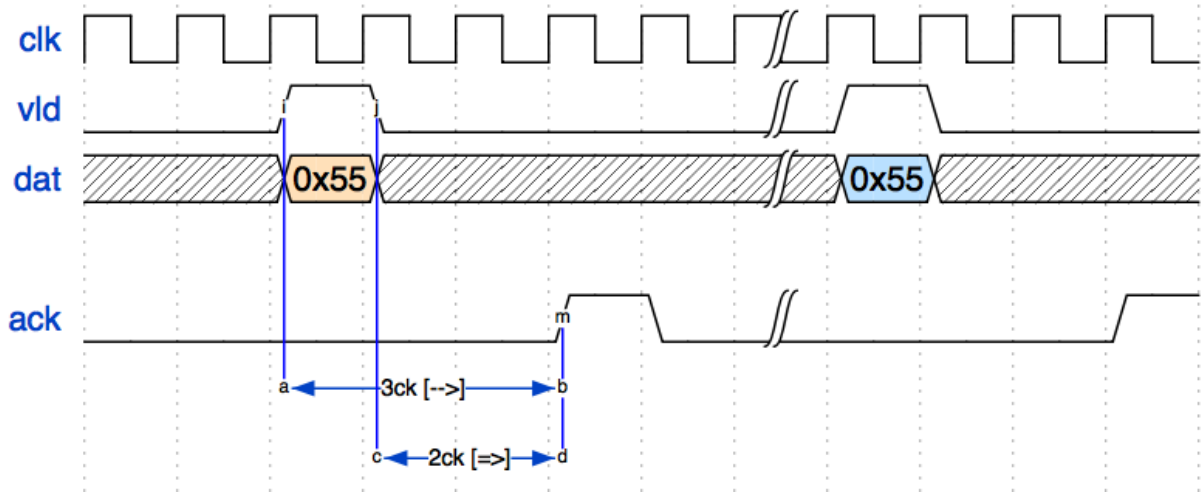

Figure 5 Implication Operator example

JasperGold Coverage:

The JasperGold® Coverage App offers coverage data within a formal proof setting to reinforce confidence in results and bridge the gap with simulation coverage, aiding in achieving coverage closure. It employs branch, statement, expression, and functional code coverage models to delineate the design space under examination. Each coverage model partitions the design into elements based on the RTL constructs defined in the hardware description language (HDL). Every cover item signifies a condition of

accessibility for the construct. The accessibility assessment concentrates on cover items under specific conditions, facilitating a comparison between the user environment and the inactive code environment to observe the impacts of constraints in the proof setting.

Coverage App Flow:

Launch the Coverage App with the following command:

```
%jp -cov
```

To launch the tool with an existing database or one or more scripts, follow -cov with the database name or file names.

The general flow of the Coverage App follows:

1. Initialize the Coverage App and create the coverage model.

2. Analyze and elaborate the design.

3. Specify the global clocks and reset.

4. Add and remove targets.

5. Check reachability.

Code used for Coverage:

Synchronous FIFO:

```
module syncFIFO_v2
          #(parameter WIDTH = 4,
            parameter DEPTH_LEN = 4) // 2^4 depth
          (
          i_clk, i_rst_n, i_data, wr_en,
          rd_en, o_data, o_full, o_empty
          );
  input i_clk, i_rst_n;
  input [WIDTH-1:0] i_data;
  input wr_en, rd_en;
  output  reg[WIDTH-1:0] o_data;

  reg [WIDTH-1:0] mem [(1<<DEPTH_LEN)-1:0];
```

```
  output  o_full;
  output  o_empty;

  // points to the address to read/write to
  // notice extra bit
  reg [DEPTH_LEN: 0] rd_ptr, wr_ptr;

  wire rd_req, wr_req;
  assign rd_req = rd_en && !o_empty;
  assign wr_req = wr_en && !o_full;
  wire [DEPTH_LEN: 0] fill;
  assign fill = (wr_ptr - rd_ptr);
  assign o_empty = (fill==0);
  assign o_full = (fill == {1'b1, {DEPTH_LEN{1'b0}}});
  always@(posedge i_clk, negedge i_rst_n)
  begin
    if(!i_rst_n)
    begin
      wr_ptr <= 5'h00;
    end

    else
    begin
      if(wr_req)
      begin
        mem[wr_ptr[DEPTH_LEN-1: 0]] <= i_data;
        wr_ptr <= wr_ptr + 1'b1;
      end
    end
  end

  always@(posedge i_clk, negedge i_rst_n)
  begin
    if(!i_rst_n)
    begin
      rd_ptr <= 5'h00;
    end
    else
    begin
      if (rd_req)
      begin
        rd_ptr <= rd_ptr + 1'b1;
      end
    end
  end

  // combinational read
  assign o_data = mem[rd_ptr[DEPTH_LEN-1: 0]];

  /*
  * ASSERTIONS Synchronous FIFO
  */

  // Reset startup check //
```

```
  // need this at the very begining of the simulation //
  property async_rst_startup;
       @(posedge i_clk) !i_rst_n |-> ##1 (wr_ptr==0 && rd_ptr == 0 &&
o_empty);
  endproperty
  assert property (async_rst_startup)
       else $display("rst assertion failed at strup", $time);

  // rst check in general
  property async_rst_chk;
       @(negedge i_rst_n) 1'b1 |-> ##1 @(posedge i_clk) (wr_ptr==0 &&
rd_ptr == 0 && o_empty);
  endproperty
  assert property (async_rst_chk)
          else $display("rst assertion failed: ", $time);
// 1) check if data written to a location is the same data read when read
ptr reaches the location
  sequence rd_detect(ptr);
    ##[0:$] (rd_en && !o_empty && (rd_ptr == ptr));
  endsequence
// 2) Don't write to fifo if full: one of the two golden rule in fifo design
  property dont_write_if_full;
    //   @(posedge   i_clk)   disable   iff(!i_rst_n)   o_full   |->   ##1
$stable(wr_ptr);
    // alternative way of writing the same assertion
       @(posedge i_clk) disable iff(!i_rst_n) wr_en && o_full |-> ##1
wr_ptr == $past(wr_ptr);
  endproperty
  assert property (dont_write_if_full)
    else $display("failed at time p2: ", $time);

    // 3) don't read when empty: second golden rule
    property dont_read_if_empty;
         @(posedge i_clk) disable iff(!i_rst_n) rd_en && o_empty |-> ##1
$stable(rd_ptr);
    endproperty

    assert property (dont_read_if_empty)
      else $display("failed at time p3: ", $time);

     // 4) rd/wr ptr should onlu increment by 1 on rd/wr req
     property inc_wr_one;
      @(posedge i_clk) disable iff(!i_rst_n) wr_en && !o_full |-> ##1
(wr_ptr-1'b1 == $past(wr_ptr));
     endproperty

     assert property (inc_wr_one)
       else $display("time p4: ", $time);

      // 5) rd/wr ptr should onlu increment by 1 on rd/wr req
      property inc_rd_ptr;
        @(posedge i_clk) disable iff(!i_rst_n) rd_en && !o_empty |-> ##1
(rd_ptr - 1'b1 == $past(rd_ptr));
      endproperty
```

```
    assert property (inc_rd_ptr)
      else $display("time p5: ", $time);

endmodule
```

TCL File:

```
#For Clearing all previous settings
clear -all

#For coverage purpose
check_cov -init -type all -model {branch toggle statement} -
toggle_ports_only
analyze -sv12 sync_fifo.sva

elaborate -top syncFIFO_v2 -parameter WIDTH 4 -parameter DEPTH_LEN 4 -
create_related_covers {precondition witness}
clock i_clk
reset i_rst_n

check_cov -measure -type {coi stimuli proof bound} -time_limit 60s -bg
```

Coverage:

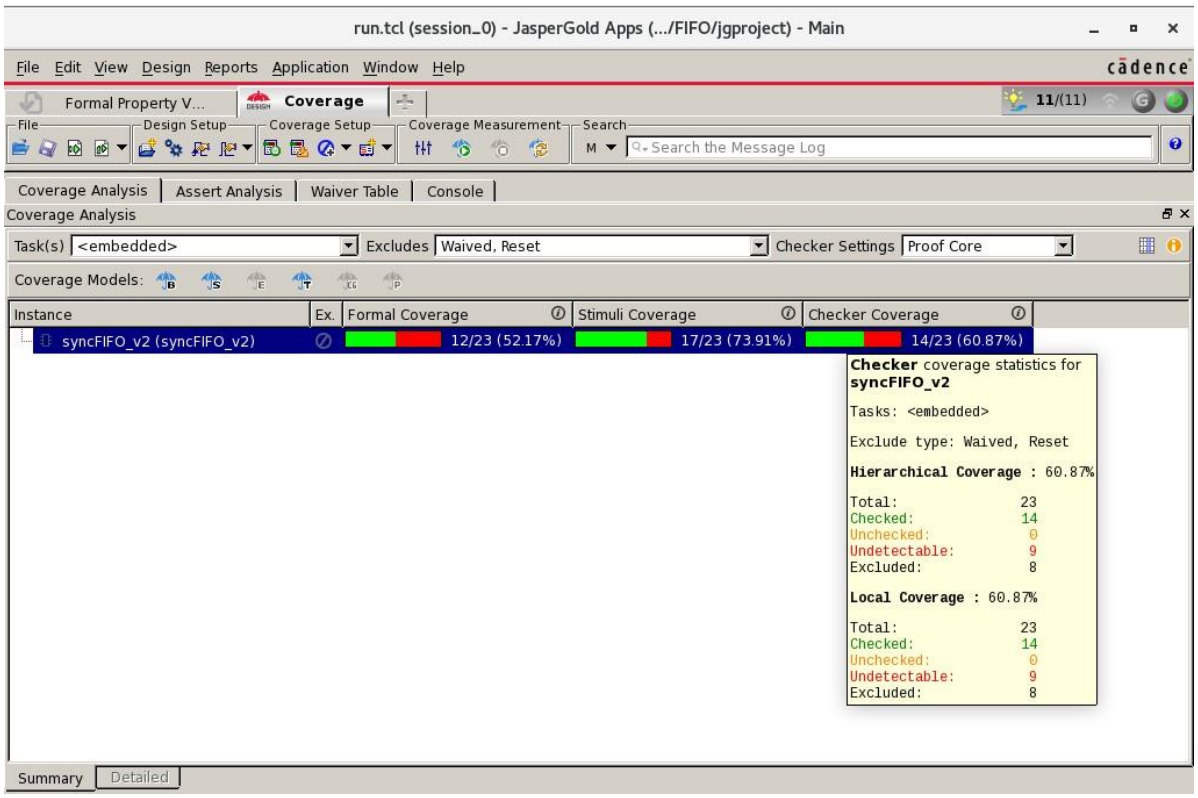

Figure 6 Formal Coverage Statistics (I)

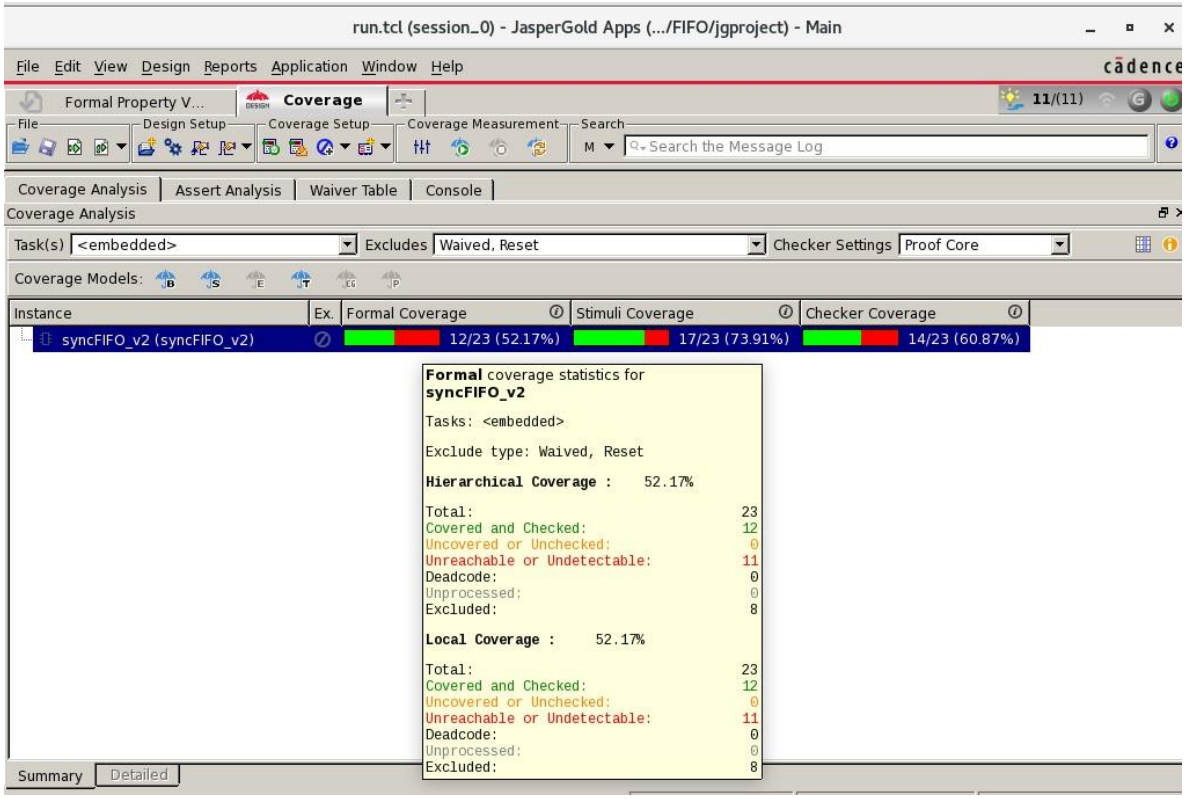

Figure 7 Formal Coverage Statistics (II)

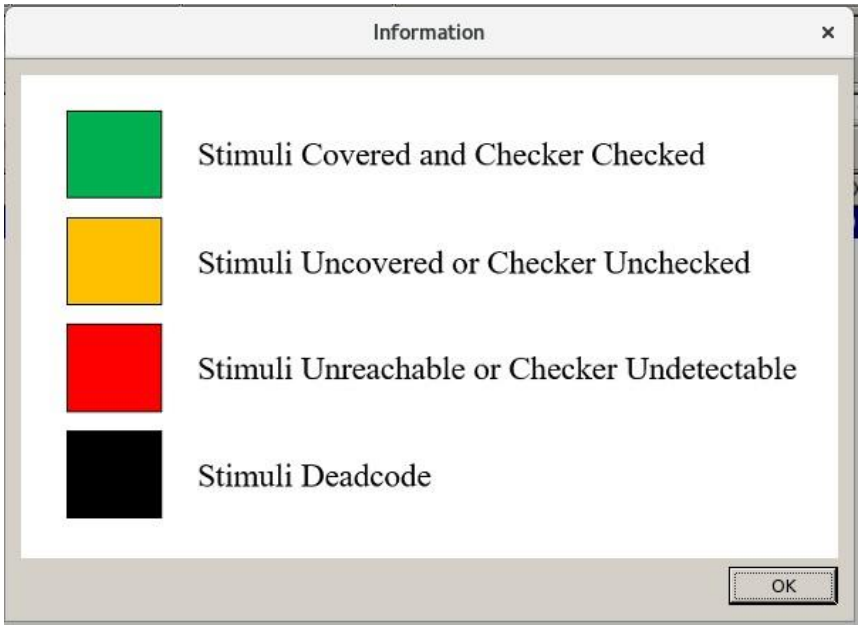

Figure 8 Formal Coverage Description

Conclusion:

In conclusion, this report covered the concept of assertion-based verification using the JasperGold tool and its application to a synchronous FIFO design. Assertion-based verification allows designers to specify intended design behaviour using assertions written in SVA. Different types of assertions like immediate assertions and concurrent assertions were explained along with their syntax.

The report provided the Verilog code for a parametrized synchronous FIFO design along with several SVA properties to verify critical aspects of its functionality. These included checking the reset behaviour, preventing writes when the FIFO is full, preventing reads when empty, and ensuring the read and write pointers increment correctly.

The JasperGold Coverage App was utilized to formally verify the FIFO and measure code coverage metrics like branch, statement, expression, and functional coverage. The coverage results and statistics were presented, demonstrating how formal tools can provide a deeper analysis of design behaviour compared to traditional simulation-based verification.

Overall, the report highlighted the benefits of assertion-based verification using formal tools like JasperGold for comprehensively verifying digital circuit designs against specified assertions and achieving thorough code coverage. This approach can improve design quality and catch corner-case bugs that may be missed by simulation alone.