**Synchronous FIFO**

**UVM Project**

**Table of contents**

**section ………………………………………………………………………. page**

Specs ……………………………………………………………………………. 3

Verification plan …………………………………………………………… 4

Bugs report …………………………………………………………………. 5

Do file …………………………………………………………………………. 5

UVM Hierarchy ……………………………………………………………. 6

Code snippets ……………………………………………………………… 8

Simulation ………………………………………………………………….. 26

Functional coverage ………..………………………………………….. 29

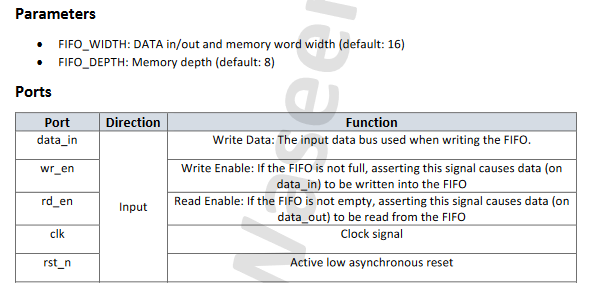
Code coverage ..………………………………………………………….. 33

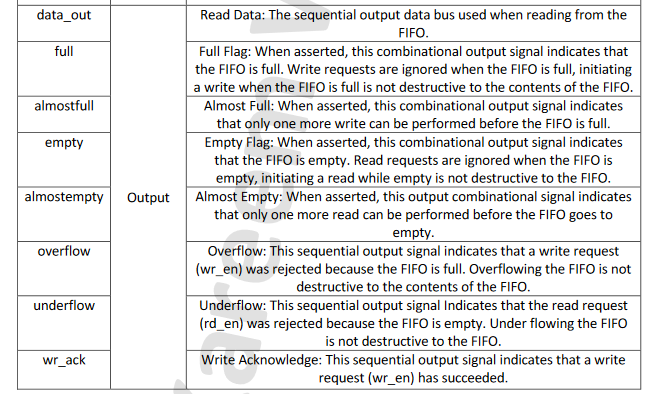
Assertion coverage ………………………………………………………. 35

UVM report ………………………………………………………………… 40

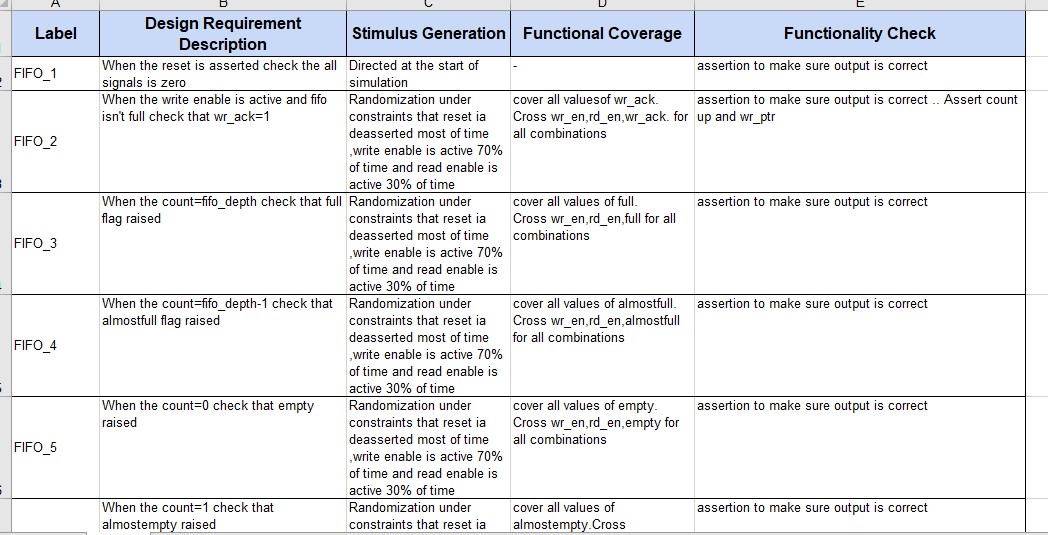
Assertion details ………………………………………………………….. 41

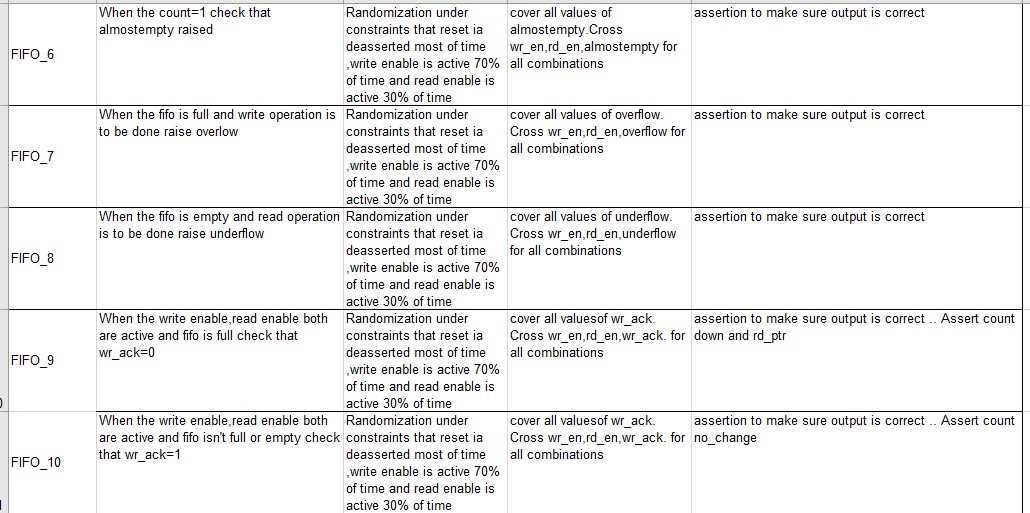
**Specs**

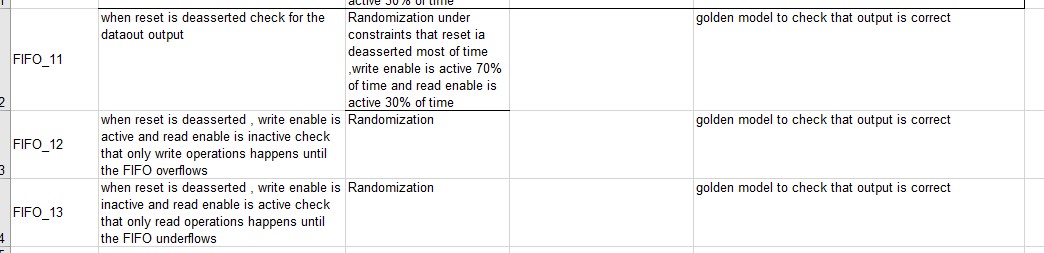
****

****

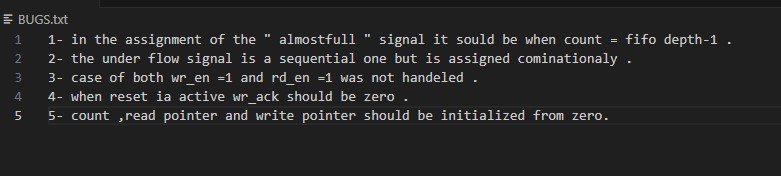
**Verification plan**



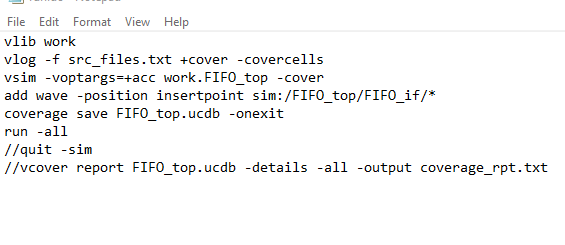


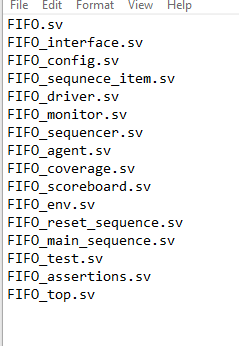


**Bugs found**

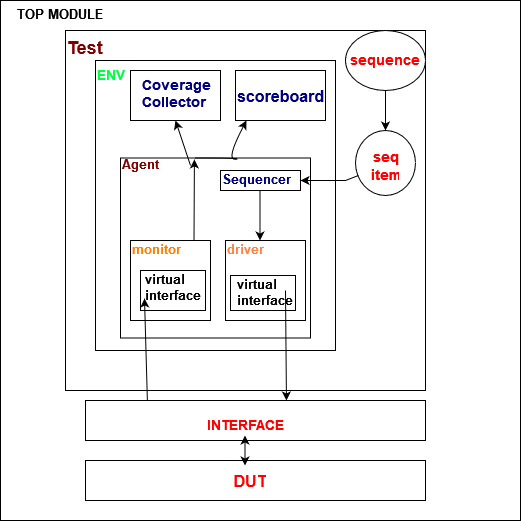


**Dofile**

****

****

**UVM Hierarchy**



**Top module**

This is where the execution starts .

This block is responsible for generating clock , instantiating the dut , binding assertions , setting the virtual interface into the configuration database and running the test .

**UVM TEST**

Builds the environment and sequences , getting the virtual interface from database , setting the configuration object and running sequences to the sequencer .

**UVM SEQUENCE**

Sequences represents the stimulus we generate to test the dut through generating several sequence items .

**UVM SEQUENCE ITEMS**

These are the data we use to drive the dut and used to generate random sequence stimulus.

**UVM SEQUENCER**

This is a fifo that registers the generated sequence items to push them to the driver

**UVM ENVIRONMENT**

Builds and connects the agents and analysis components (coverage collector and scoreboard)

**UVM DRIVER**

Pulls the registered items from the sequencer and then assigning them to the virtual interface that talks directly with the dut

**UVM AGENT**

Builds the monitor , driver and sequencer .

Connects the driver to the sequencer .

Gets the config object from data base to assign its virtual interface to ones of driver and monitor .

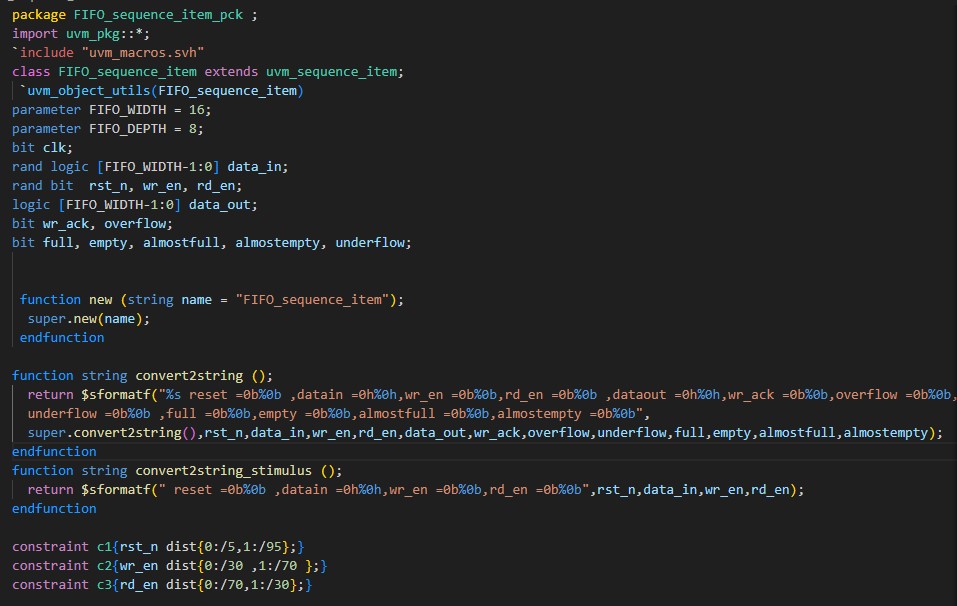
**UVM ANALYSIS COMPONENTS**

**Scoreboard :** receives a sequence item from the monitor and compares it with the reference model to check functionality .

**Coverage collector** : receives a sequence item from the monitor and samples the data for functional coverage .

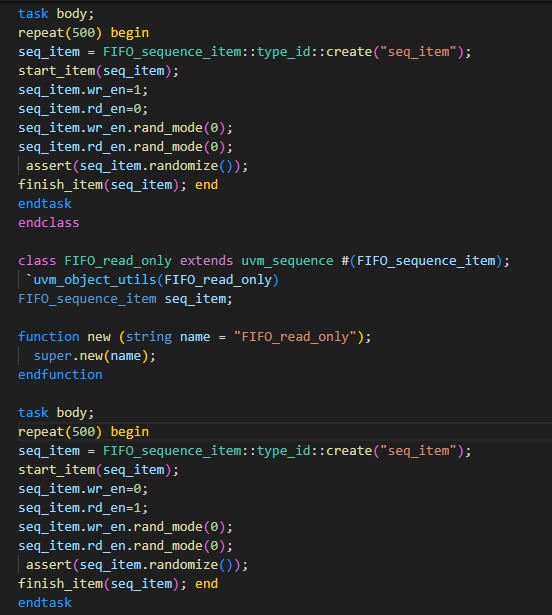
**Snippets**

**Sequence item**

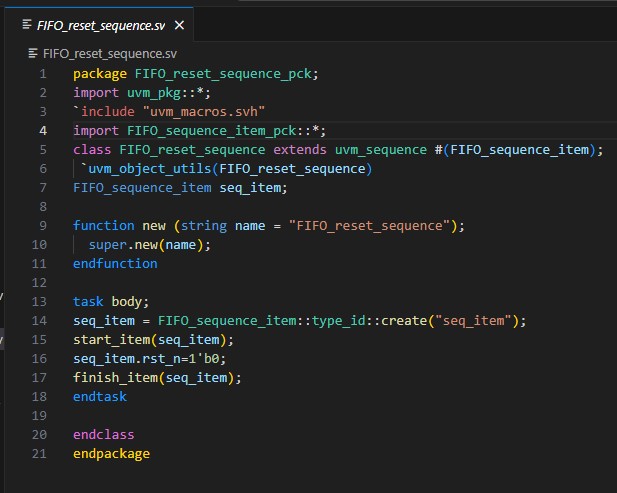


**Sequence**

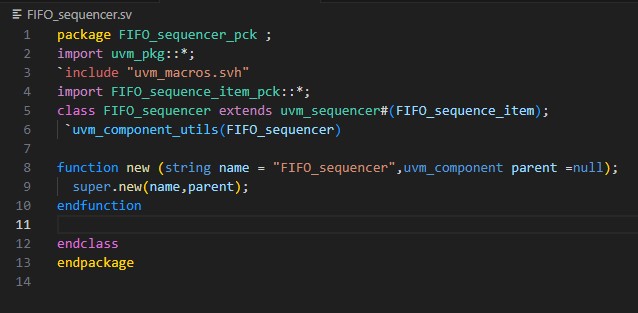




**Reset**



**Sequencer**



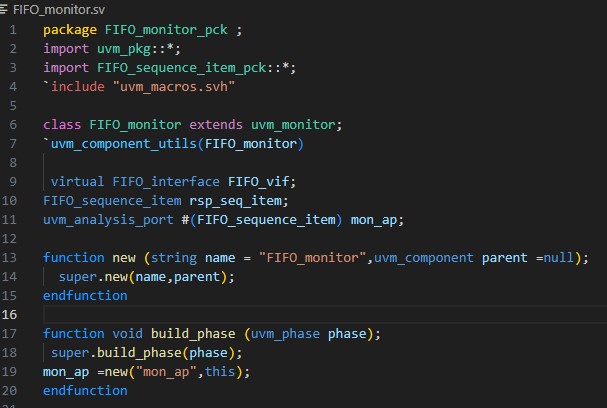
**Driver**

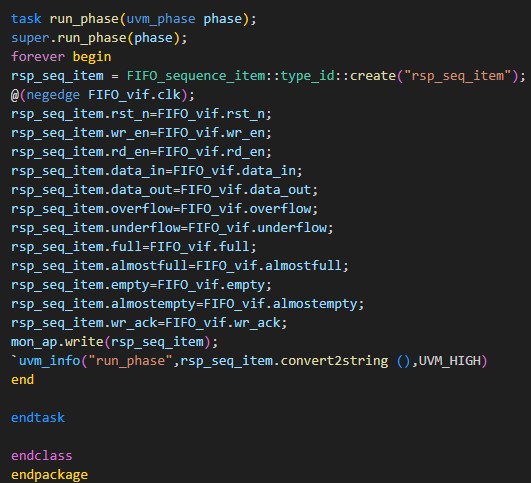
****

**Configuration**



**Monitor**

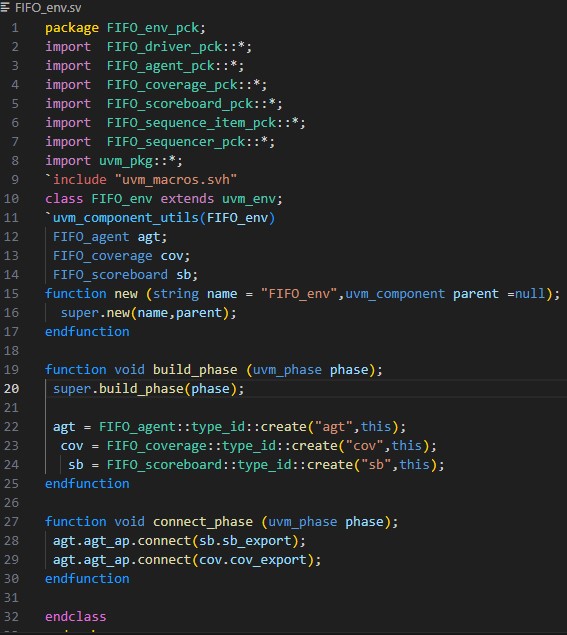




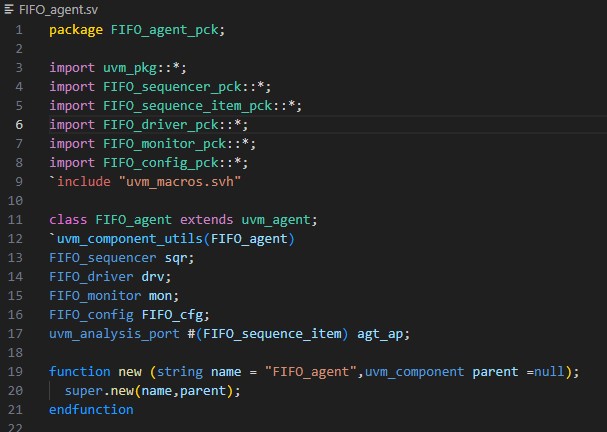
**Interface**

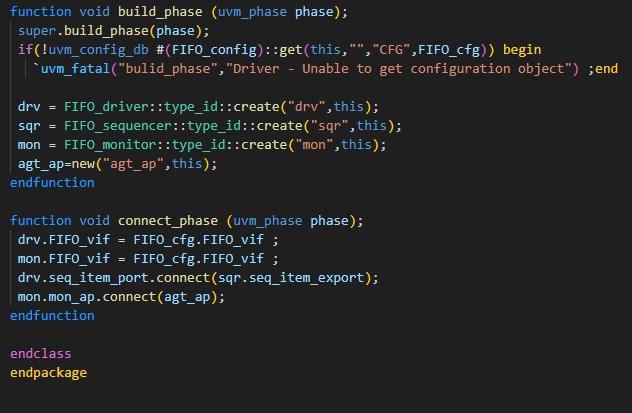


**Environment**



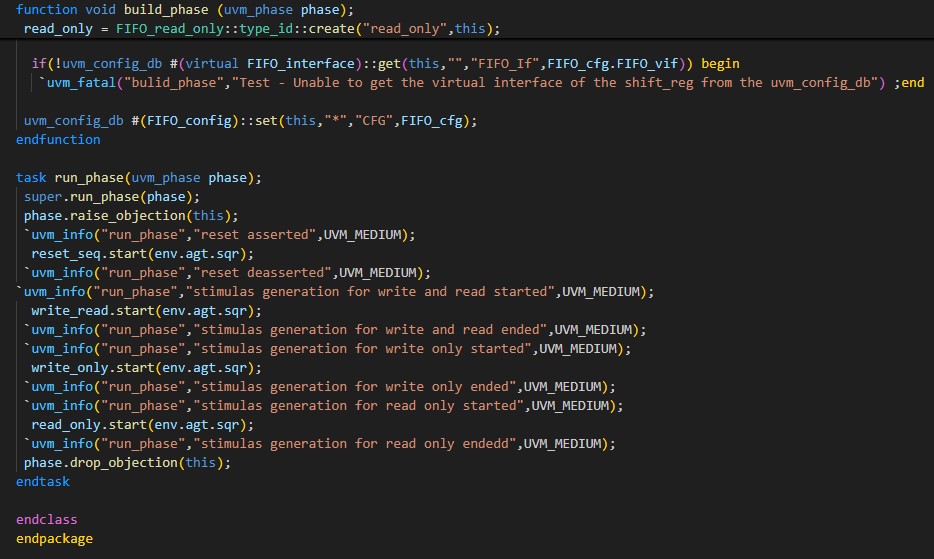
**Agent**





**Test**

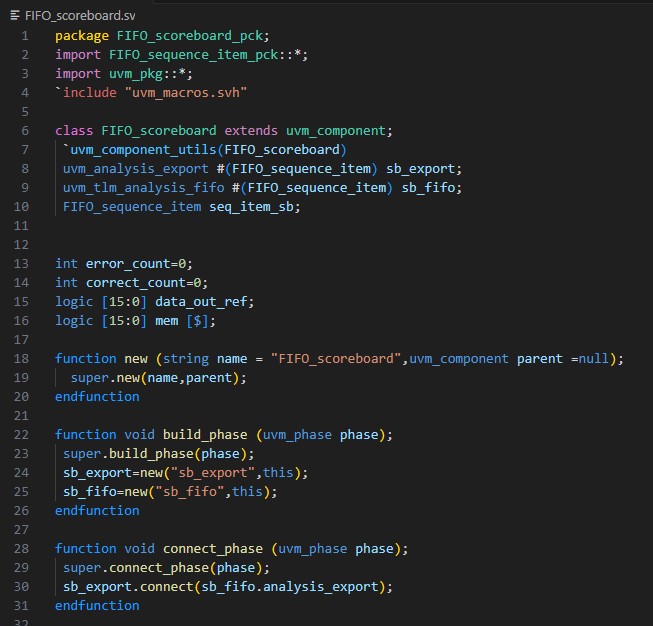


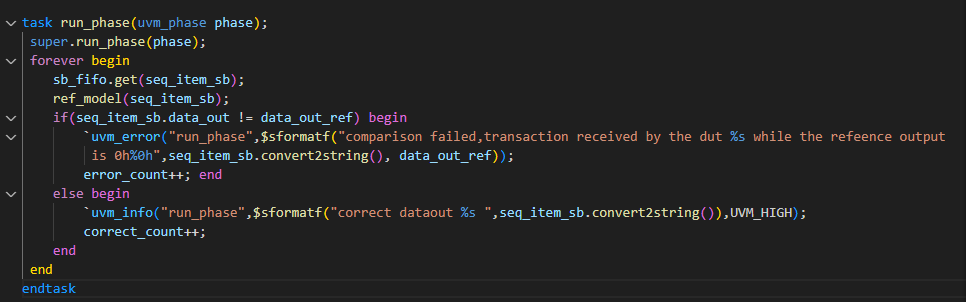


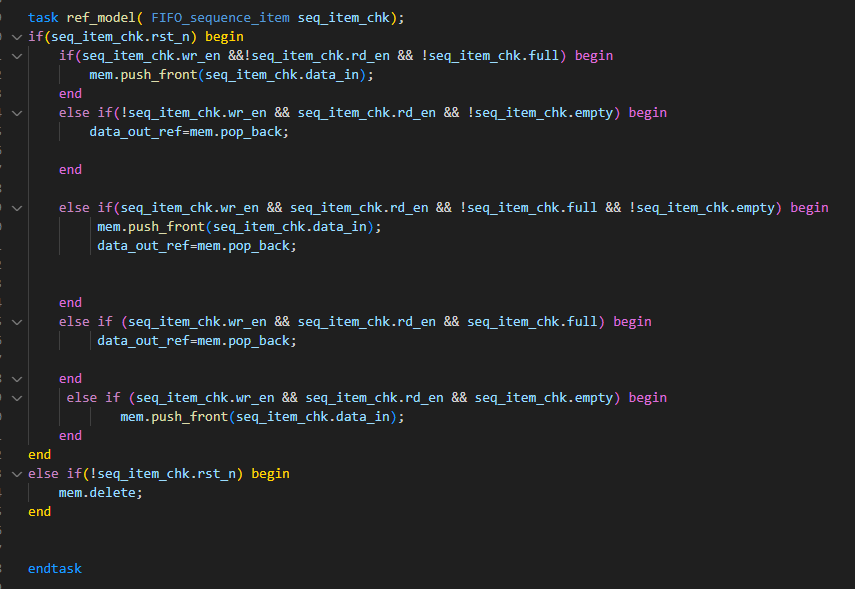
**Top**



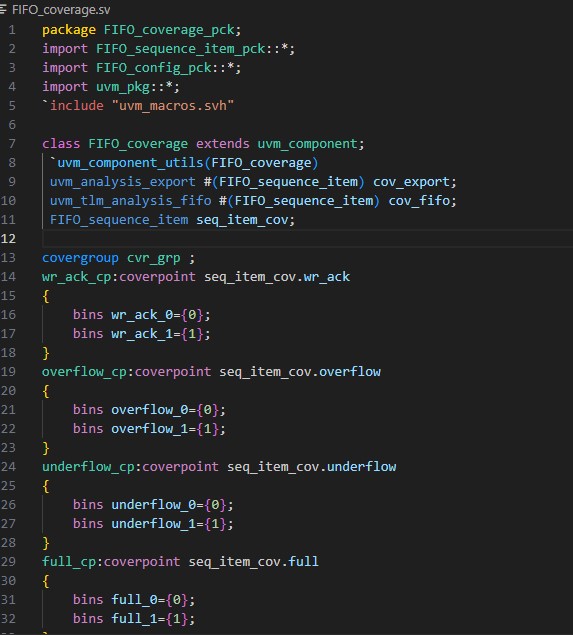
**Scoreboard**

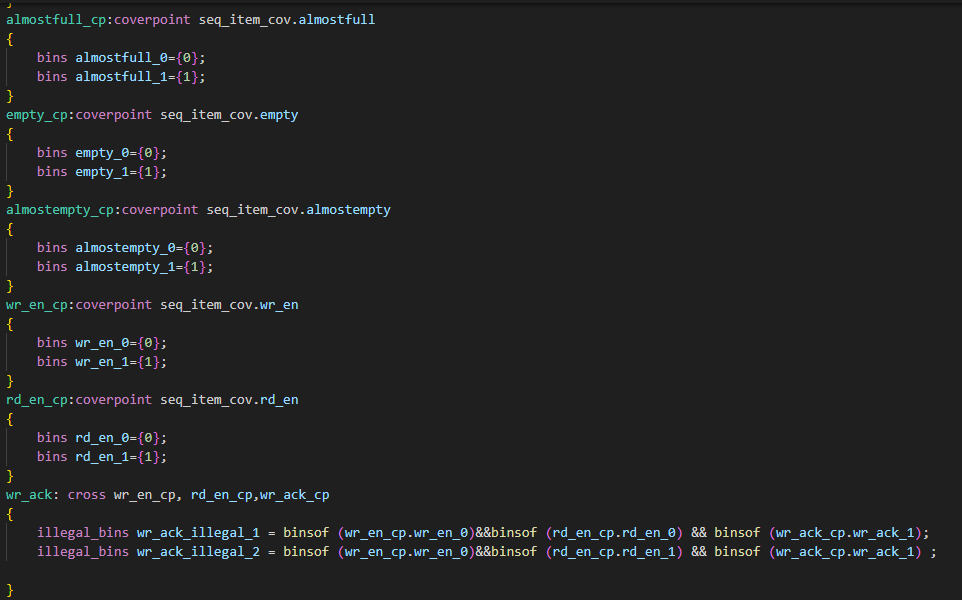


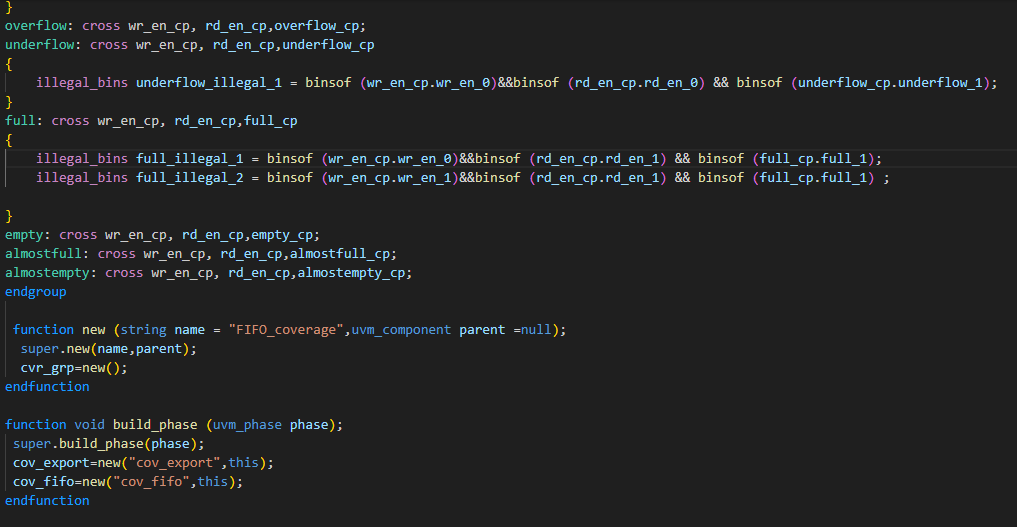
****

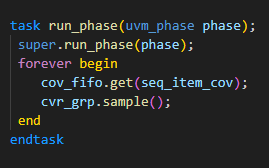
****

**Coverage collector**



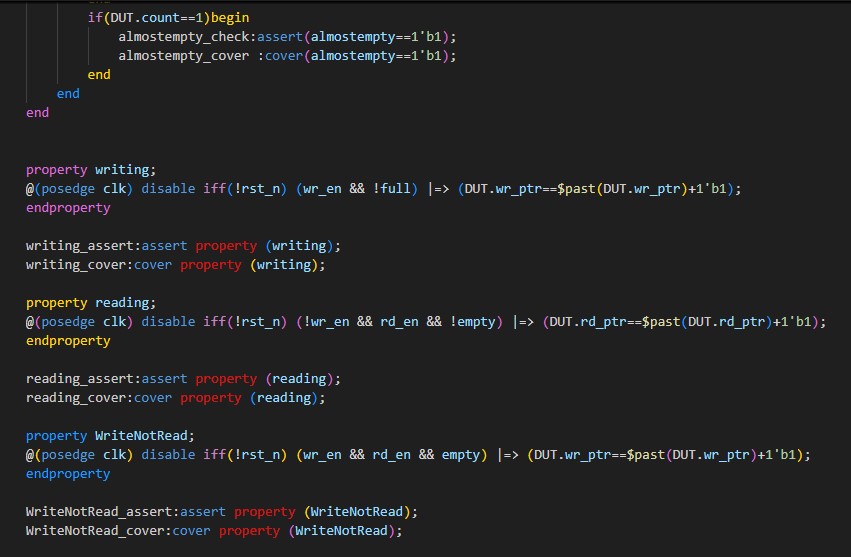
****

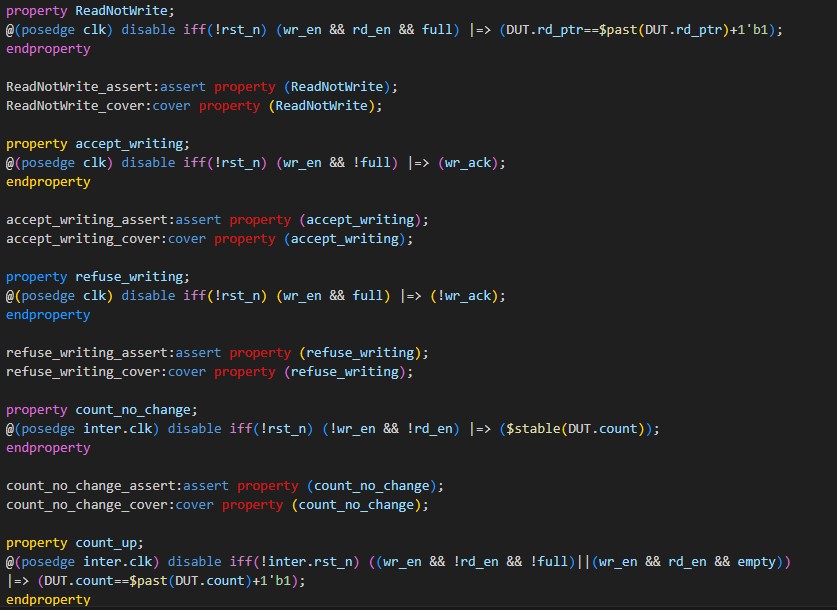
****

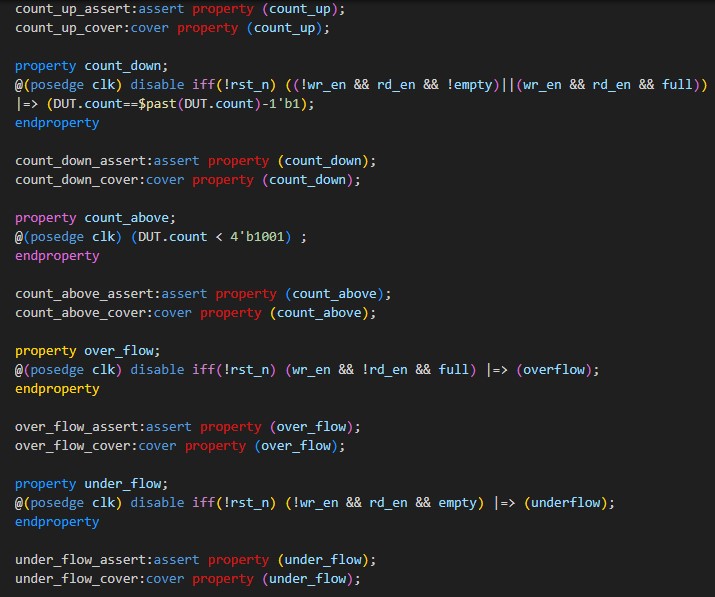
****

**Assertions**

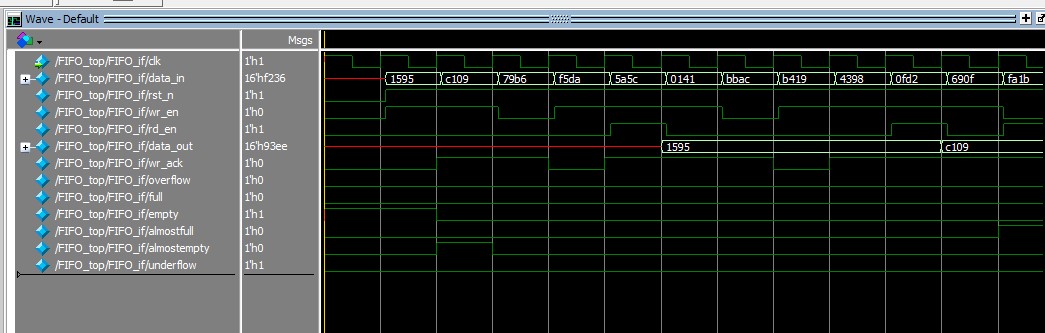


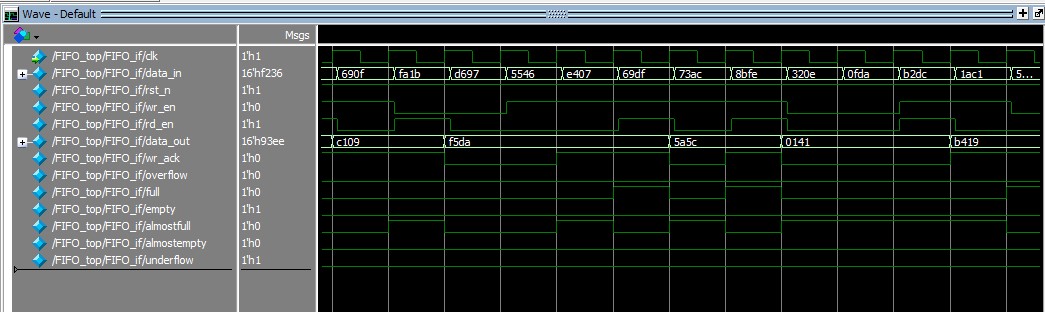


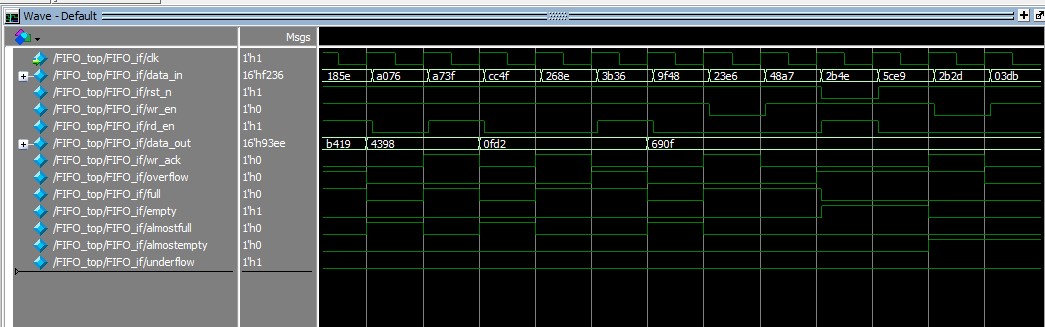


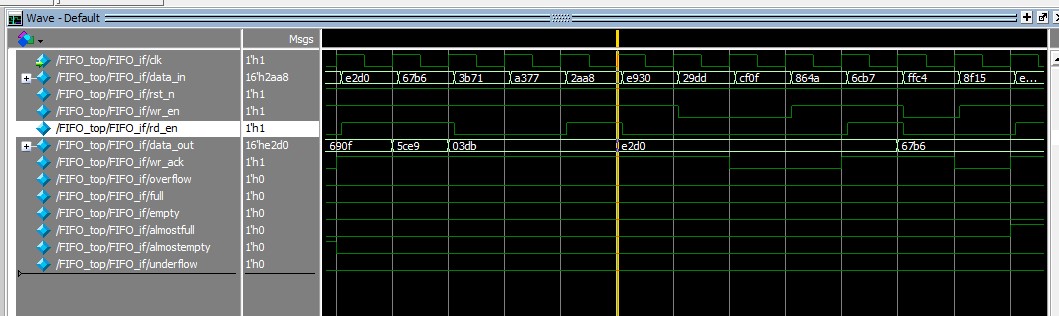


**Simulation(read and write)**

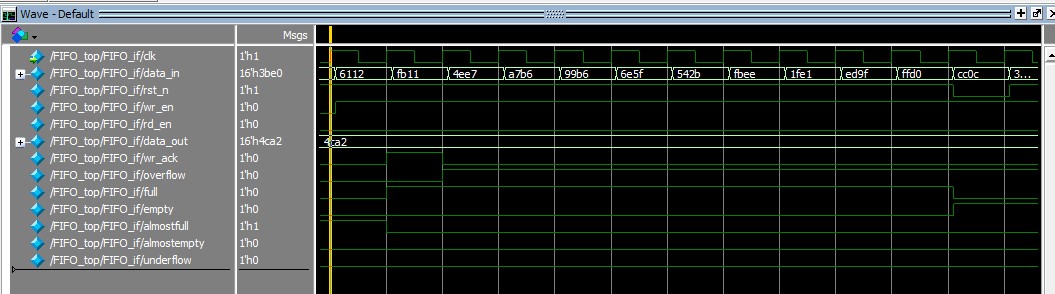
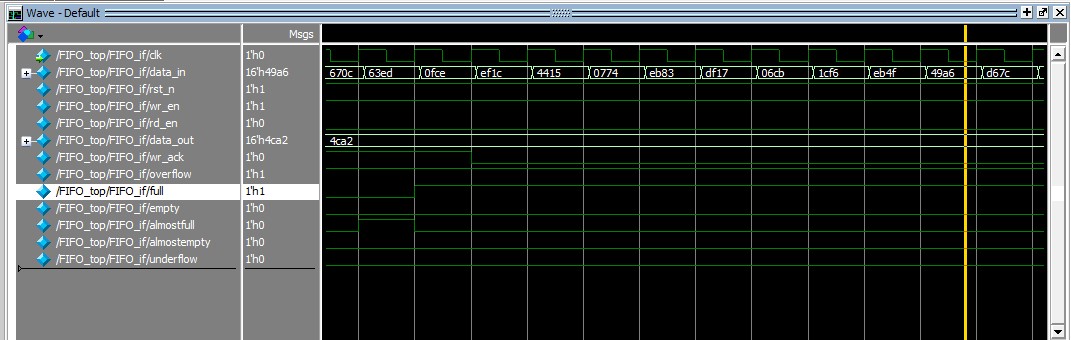




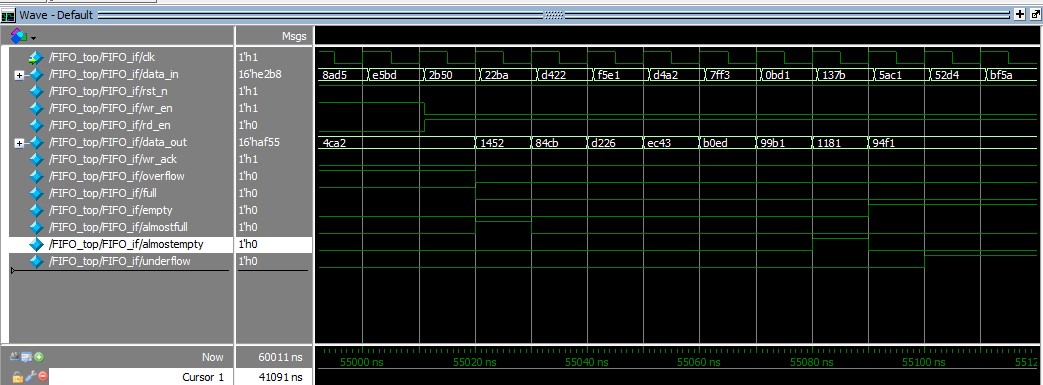




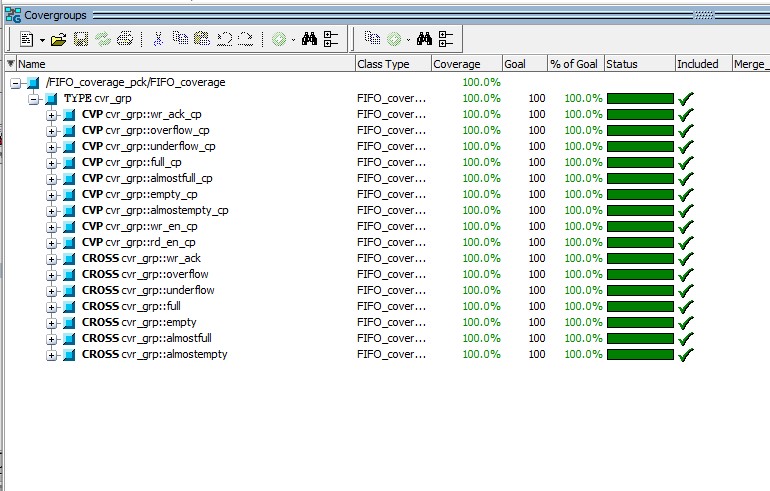
**Wrire only sequence**

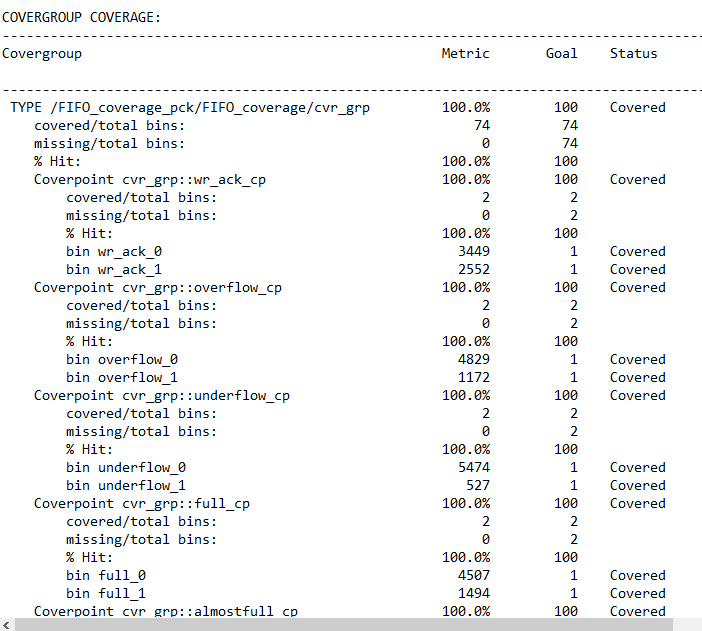


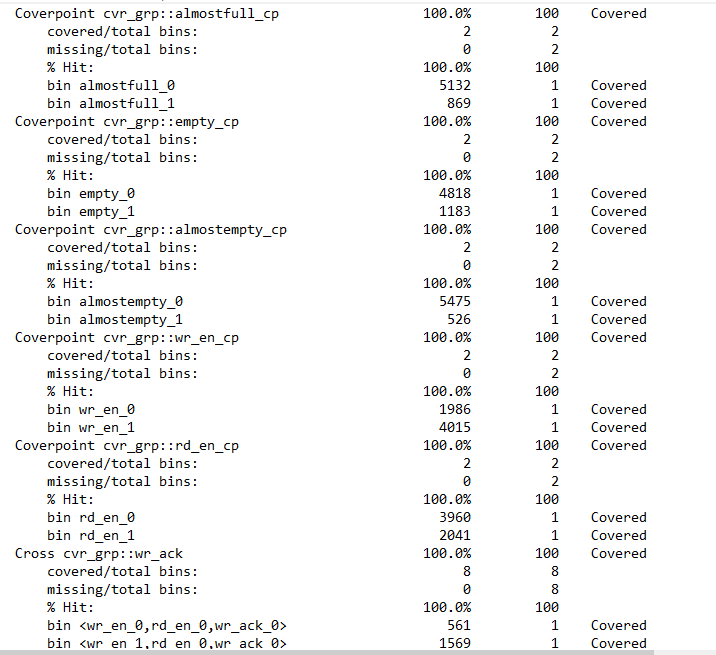
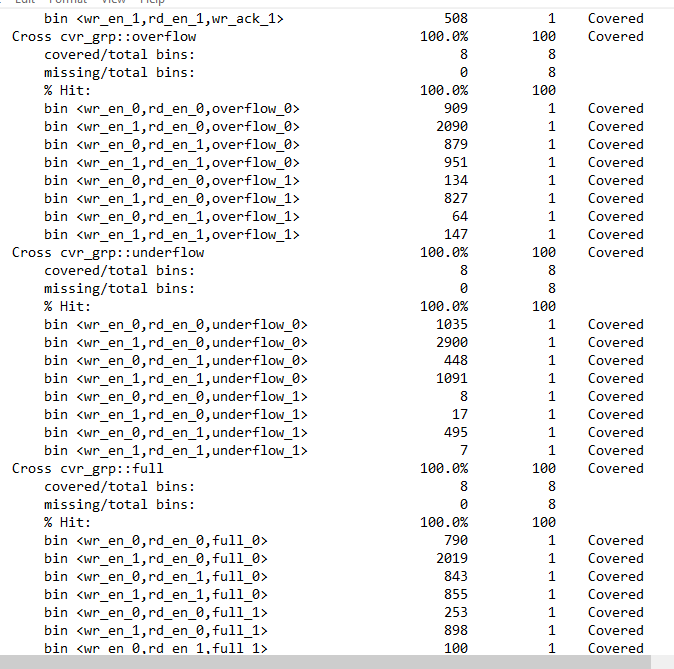
**Read only**

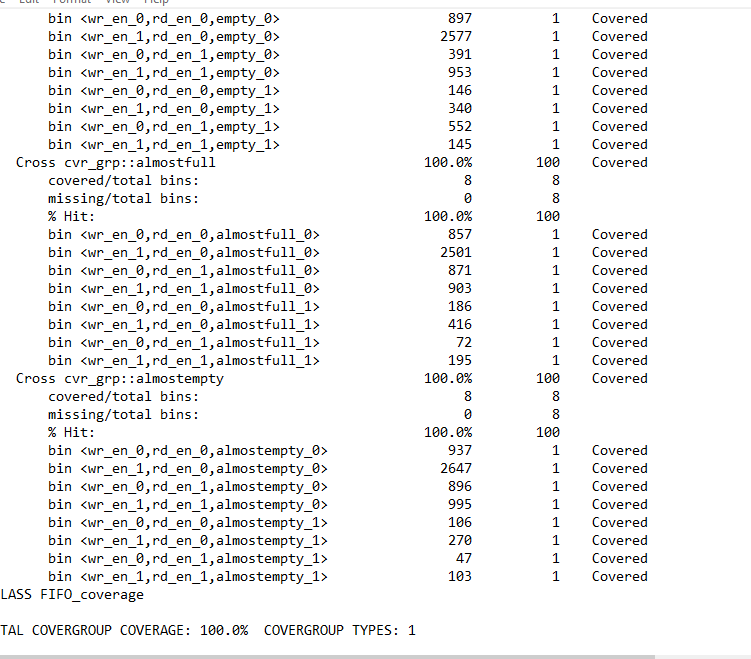


**Functional coverage**

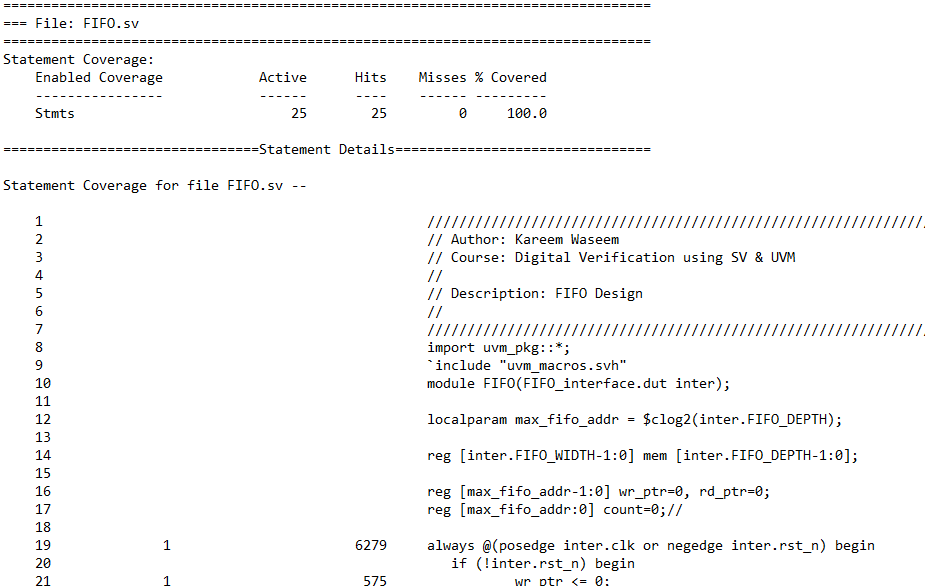


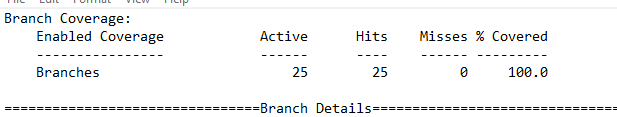
****

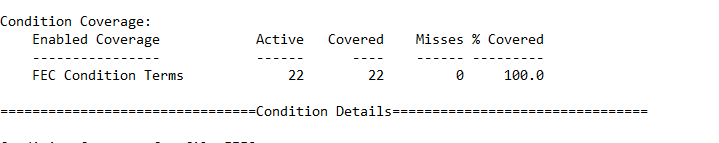
**** ****

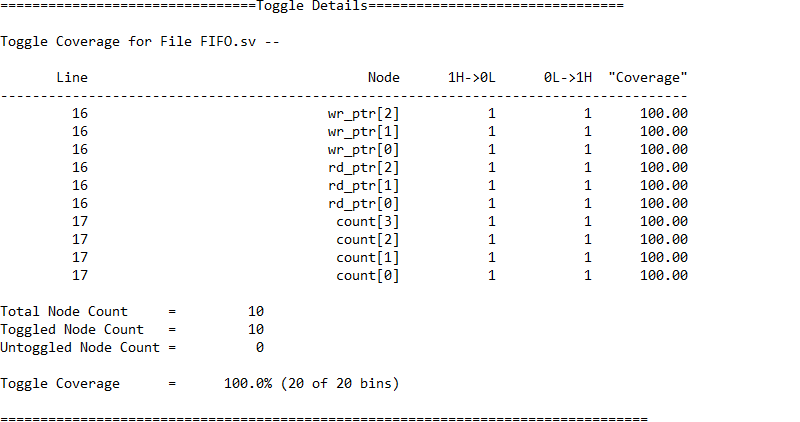
****

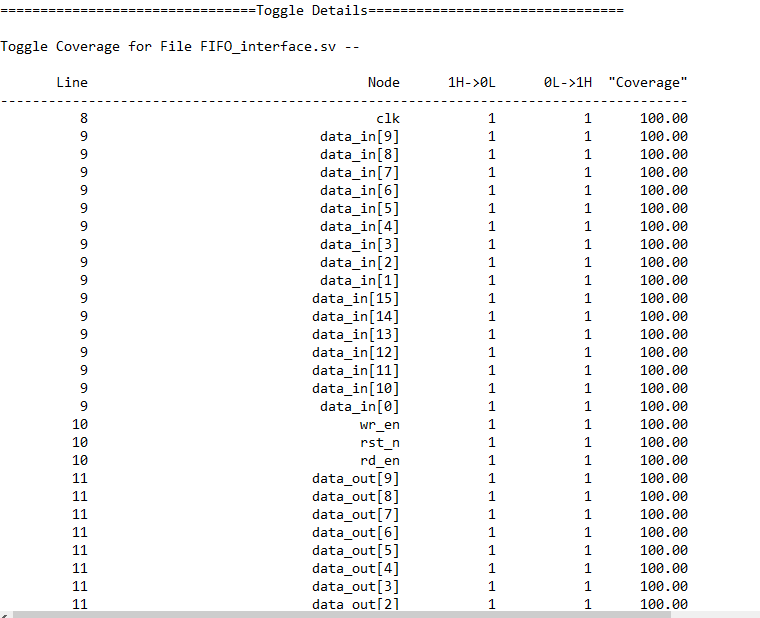
**Code coverage**

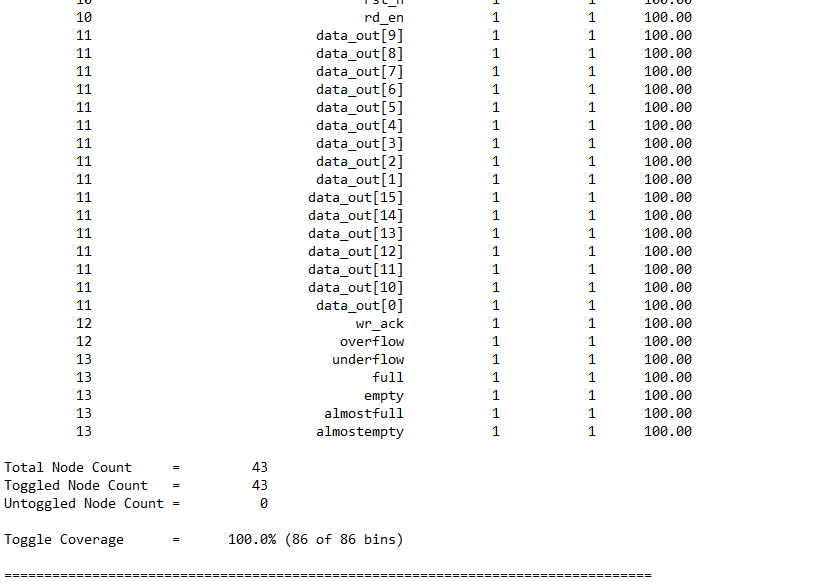
****

****

****

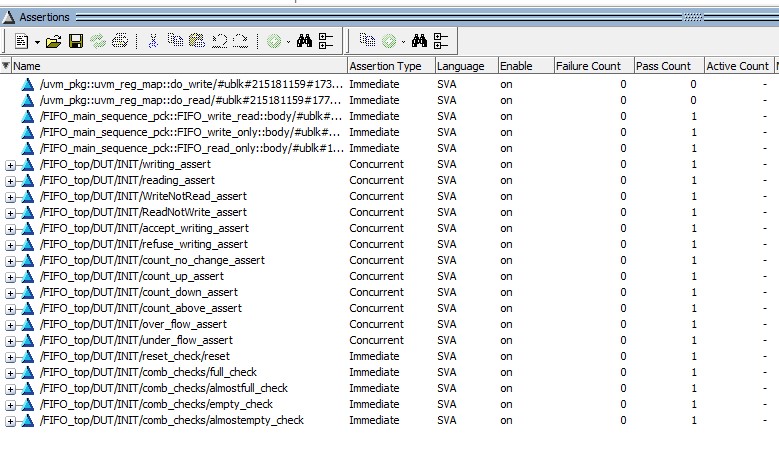
****

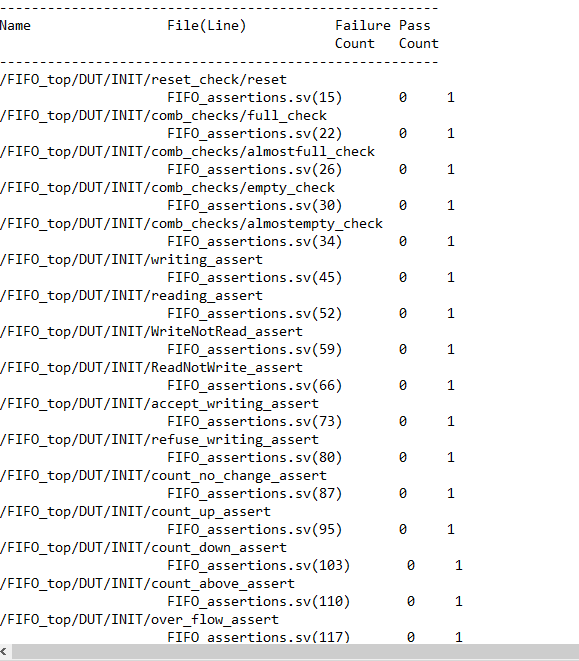
****

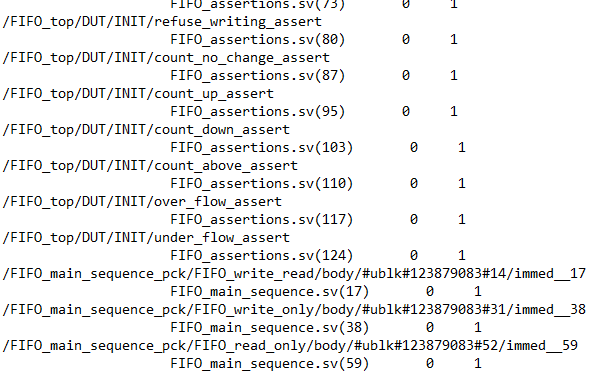
****

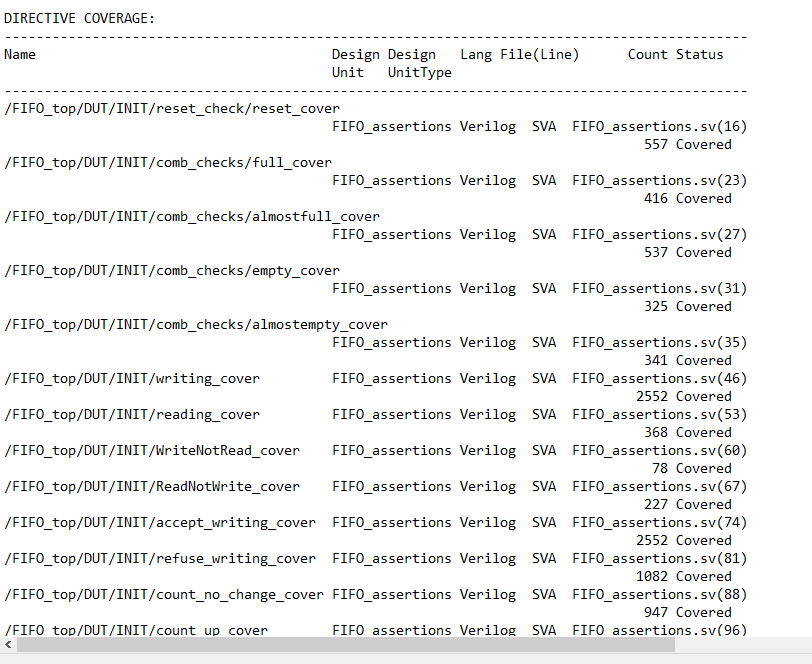
**Assertion coverage**





****

****

****

****

**UVM report**



**Assertion details**

**Immediate Assertions**

|  |  |
| --- | --- |
| Feature | Assertion |
| When reset is asserted all internal signals and outputs should be zero except the empty flag and data out | if(!rst\_n)  reset: assert final (DUT.count==0 && DUT.rd\_ptr==0 && DUT.wr\_ptr==0 && empty && !full && !almostfull && !almostempty); |
| When reset is deasserted and count=FIFO\_DEPTH ,the full flag rises. | if(rst\_n) begin  if(DUT.count==FIFO\_DEPTH)  full\_check:assert(full==1'b1); |
| When reset is deasserted and count=FIFO\_DEPTH-1 ,the almostfull flag rises. | if(rst\_n) begin  if(DUT.count==FIFO\_DEPTH-1)  full\_check:assert(almostfull==1'b1); |
| When reset is deasserted and count= 1 ,the almostempty flag rises. | if(rst\_n) begin  if(DUT.count==1'b1)  full\_check:assert(almostempty ==1'b1); |
| When reset is deasserted and count= 0 ,the empty flag rises. | if(rst\_n) begin  if(DUT.count==1'b0)  full\_check:assert(empty ==1'b1); |

**Concurrent Assertions**

|  |  |
| --- | --- |
| Feature | Assertion |
| At posedge clk if write enable is high and fifo isn’t full the write pointer increments in a cycle | @(posedge clk) disable iff(!rst\_n) (wr\_en && !full) |=> (DUT.wr\_ptr==$past(DUT.wr\_ptr)+1'b1); |
| At posedge clk if write enable is low,read enable is high and fifo isn’t empty the read pointer increments in a cycle | @(posedge clk) disable iff(!rst\_n) (!wr\_en && rd\_en && !empty) |=> (DUT.rd\_ptr==$past(DUT.rd\_ptr)+1'b1); |
|  |  |
| At posedge clk if write enable is high,read enable is high and fifo is empty the write pointer increments in a cycle | @(posedge clk) disable iff(!rst\_n) (wr\_en && rd\_en && empty) |=> (DUT.wr\_ptr==$past(DUT.wr\_ptr)+1'b1); |
| At posedge clk if write enable is high,read enable is high and fifo is full the read pointer increments in a cycle | @(posedge clk) disable iff(!rst\_n) (wr\_en && rd\_en && full) |=> (DUT.rd\_ptr==$past(DUT.rd\_ptr)+1'b1); |
| At posedge clk if write enable is high and fifo is full the wr\_ack will be low in a cycle(failing to write) | @(posedge clk) disable iff(!rst\_n) (wr\_en && full) |=> (!wr\_ack); |
| At posedge clk if write enable is high and fifo isn’t full the wr\_ack will be high in a cycle(manage to write) | @(posedge clk) disable iff(!rst\_n) (wr\_en && !full) |=> (wr\_ack); |
| At posedge clk if write enable is low,read enable is low the count will not change | @(posedge inter.clk) disable iff(!rst\_n) (!wr\_en && !rd\_en) |=> ($stable(DUT.count)); |
| At posedge clk if write enable is high,read enable is low and fifo isn’t full or write enable is high,read enable is high and fifo is empty the counter increments in a cycle | @(posedge inter.clk) disable iff(!inter.rst\_n) ((wr\_en && !rd\_en && !full)||(wr\_en && rd\_en && empty))  |=> (DUT.count==$past(DUT.count)+1'b1); |
| At posedge clk if write enable is high,read enable is high and fifo is full or write enable is low,read enable is high and fifo isn’t empty the counter decrements in a cycle | @(posedge clk) disable iff(!rst\_n) ((!wr\_en && rd\_en && !empty)||(wr\_en && rd\_en && full))  |=> (DUT.count==$past(DUT.count)-1'b1); |
| Count shouldn’t exceed the fifo depth at any time | @(posedge clk) (DUT.count < 4'b1001) ; |
| At posedge clk if write enable is high,read enable is low and fifo is full the overflow rises in a cycle | @(posedge clk) disable iff(!rst\_n) (wr\_en && !rd\_en && full) |=> (overflow); |
| At posedge clk if write enable is low,read enable is high and fifo is empty the underflow rises in a cycle | @(posedge clk) disable iff(!rst\_n) (!wr\_en && rd\_en && empty) |=> (underflow); |
|  |  |
|  |  |

<https://github.com/MostafaMasoud288/FIFO_UVM_Project.git>

please visit this repo link for more information…….