FUNCTIONAL AND FORMAL VERIFICATION OF DIGITAL DESIGN

UE21EC343AB4

PREPARED BY:
ADI M -- ----------------PES1UG21EC012
Sujal ------------------- PES1UG21EC300
ADITYA BHARATH ---- PES1UG21EC343
B M MADHUMITHA --- PES1UG21EC902
SHARANYA SHETTY----PES1UG21EC909

GUIDED BY :
prof. Sudheendra Kumar K

TABLE OF CONTENTS

PG.NO.

1. Introduction

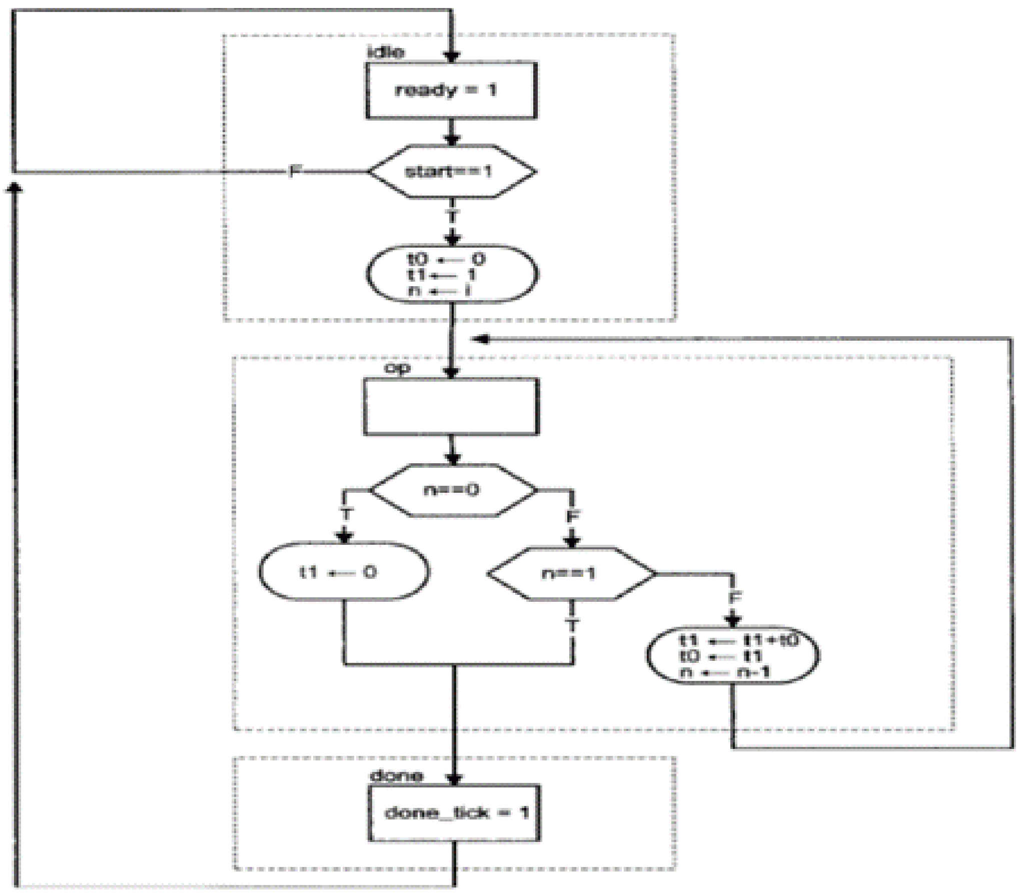
1.1 Overview of the Fibonacci series design

The Fibonacci series, a renowned mathematical sequence, is characterized by each number being the sum of the two preceding ones. Typically initiated with 0 and 1, this project delves into the intricacies of designing and verifying a digital circuit that systematically generates the Fibonacci series.

$$fib(i) = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ fib(i-1) + fib(i-2) & \text{if } i > 1 \end{cases}$$

1.2 Purpose of formal and functional verification

Formal and functional verification stand as pivotal pillars in the digital design process. Formal verification meticulously examines whether the design aligns with its specified requirements. Meanwhile, functional verification ensures that the design adheres to its intended behavior. These verification processes play a pivotal role in the early detection and rectification of errors during the design phase, ultimately mitigating the likelihood of issues in the final product.



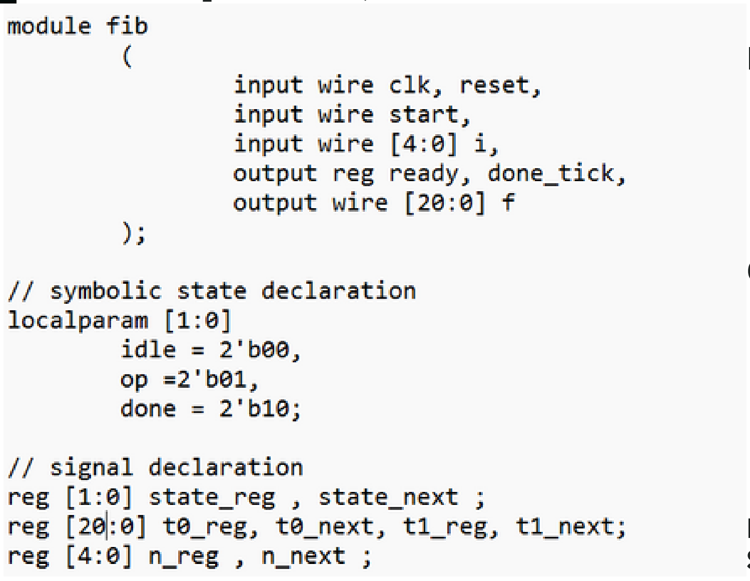
An ASMD(Algorithmic State Machine with Datapath) chart has three states.

1. Idle State:
    - The circuit is in an idle state, indicating that it is not actively computing Fibonacci numbers.
    - This state is the initial state and represents the system waiting for a ‘start’ signal.
2. Start State:
    - When the ‘start’ signal is asserted, the system transitions to the ‘op’ (operation) state.
3. Op State:
    - In the ‘op’ state, initial values are loaded into three registers: t0, t1, and n.
    - The t0 and t1 registers are loaded with 0 and 1, representing fib(0) and fib(1), respectively.
    - The n register is loaded with the desired number of iterations, denoted as "i."

2. Design Description

03

2.1 Module breakdown: Inputs, outputs, and internal logic



```
module fib
        (
                input wire clk, reset,
                input wire start,
                input wire [4:0] i,
                output reg ready, done_tick,
                output wire [20:0] f
        );

// symbolic state declaration
localparam [1:0]
        idle = 2'b00,
        op =2'b01,
        done = 2'b10;

// signal declaration
reg [1:0] state_reg , state_next ;
reg [20:0] t0_reg, t0_next, t1_reg, t1_next;
reg [4:0] n_reg , n_next ;
```

Inputs:

- input wire clk: Clock signal.
- input wire reset: Reset signal.
- input wire start: Start signal.
- input wire [4:0] i: 5-bit input for the desired number of iterations.

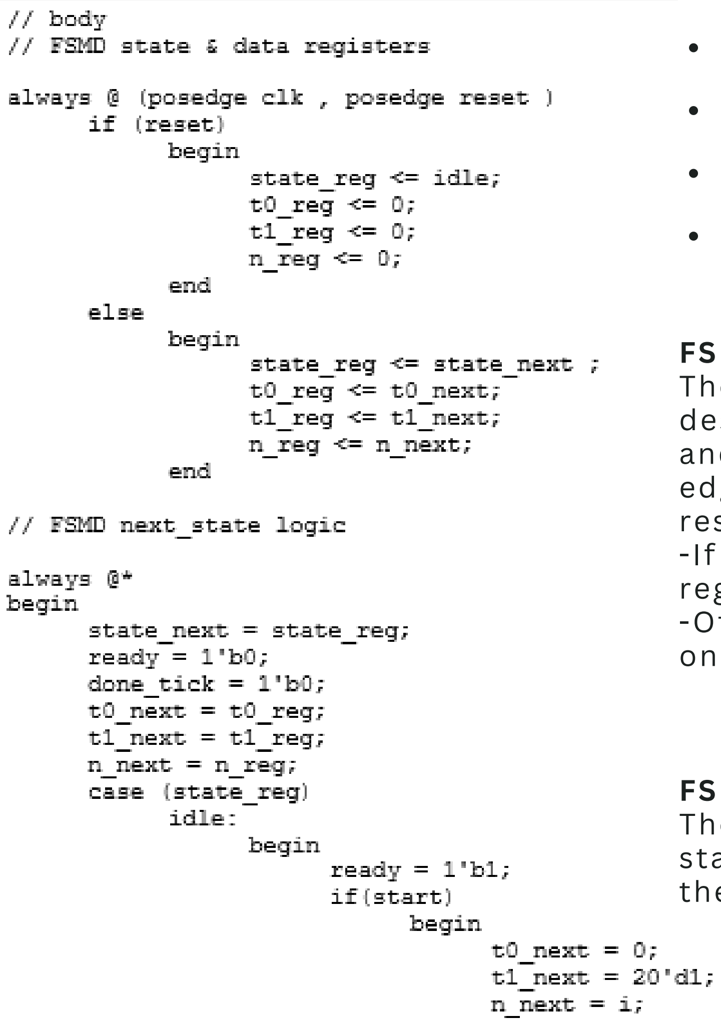
Outputs:

- output reg ready: Output indicating readiness.
- output reg done_tick: Output indicating completion.
- output wire [19:0] f: 20-bit output representing the Fibonacci series.

Internal Logic:

State Registers:

- state_reg: Register holding the current state.
- t0_reg, t1_reg: Registers holding the current values for Fibonacci numbers.
- n_reg: Register holding the current iteration count.
- state_next, t0_next, t1_next, n_next: Registers for the next state and values.



```
// body
// FSMD state & data registers

always @ (posedge clk , posedge reset )
      if (reset)
            begin
                  state_reg <= idle;
                  t0_reg <= 0;
                  t1_reg <= 0;
                  n_reg <= 0;
            end
      else
            begin
                  state_reg <= state_next ;
                  t0_reg <= t0_next;
                  t1_reg <= t1_next;
                  n_reg <= n_next;
            end

// FSMD next_state logic

always @*
begin
      state_next = state_reg;
      ready = 1'b0;
      done_tick = 1'b0;
      t0_next = t0_reg;
      t1_next = t1_reg;
      n_next = n_reg;
      case (state_reg)
            idle:
                  begin
                        ready = 1'b1;
                        if(start)
                              begin
                                    t0_next = 0;
                                    t1_next = 20'd1;
                                    n_next = i;
```

FSMD state & data registers:

The 'FSMD state & data registers' block describes how the initialization of the state and data registers occurs when a positive edge of the clock or a positive edge of the reset signal occurs.
-If reset is asserted, it initializes all registers to their initial values.
-Otherwise, it updates the registers based on their respective next values

FSMD next_state logic:

The always @* block determines the next state and updates data registers based on the current state.

```
                        state_next = op;
                    end
            end
        op:
            if (n_reg==0)
                begin
                    t1_next = 0;
                    state_next = done;

                end
            else if (n_reg==1)
                state_next = done;
            else
                begin
                    t1_next = t1_reg + t0_reg;
                    t0_next = t1_reg;
                    n_next = n_reg - 1;
                end
        done:
            begin
                done_tick = 1'b1;
                state_next = idle;
            end
        default : state_next = idle ;
    endcase
end


//output
assign f = t1_reg;

endmodule
```

Logic specific to each state:

- **Idle State (idle):**
  - Sets **ready** to 1 if **start** is asserted.
  - Initializes values if **start** is asserted.
- **Operation State (op):**
  - Computes Fibonacci numbers until the desired iteration count is reached.
- **Done State (done):**
  - Sets **done_tick** to 1.
  - Resets to the **idle** state.

Output Assignment:
Assigns the value of t1_reg to the output wire f.

2.2 State machine description

The design strategically employs a state machine to orchestrate the iterative process of Fibonacci series generation. This state machine encompasses pivotal states such as initialization, calculation, and output, orchestrating the seamless flow of the design.

- The state machine is synchronous, driven by the positive edge of the clock (posedge clk).
- The state and data registers are updated in each clock cycle based on the current state and input conditions.
- The reset signal initializes the state machine to the idle state and sets initial values for registers

2.3 Register and signal declarations

Within this intricately designed system, registers serve as repositories for intermediate values during Fibonacci calculations, while signals facilitate seamless communication between disparate modules. The meticulous declaration of registers and signals is imperative for the precise functioning of the design.

reg [1:0] state_reg, state_next;
  - Registers to hold the current and next states of the state machine.

reg [19:0] t0_reg, t0_next, t1_reg, t1_next;
  - Registers to hold the current and next values for t0 and t1.

reg [4:0] n_reg, n_next;
  - Registers to hold the current and next values for the number of iterations (n).

3.Verification Methodologies

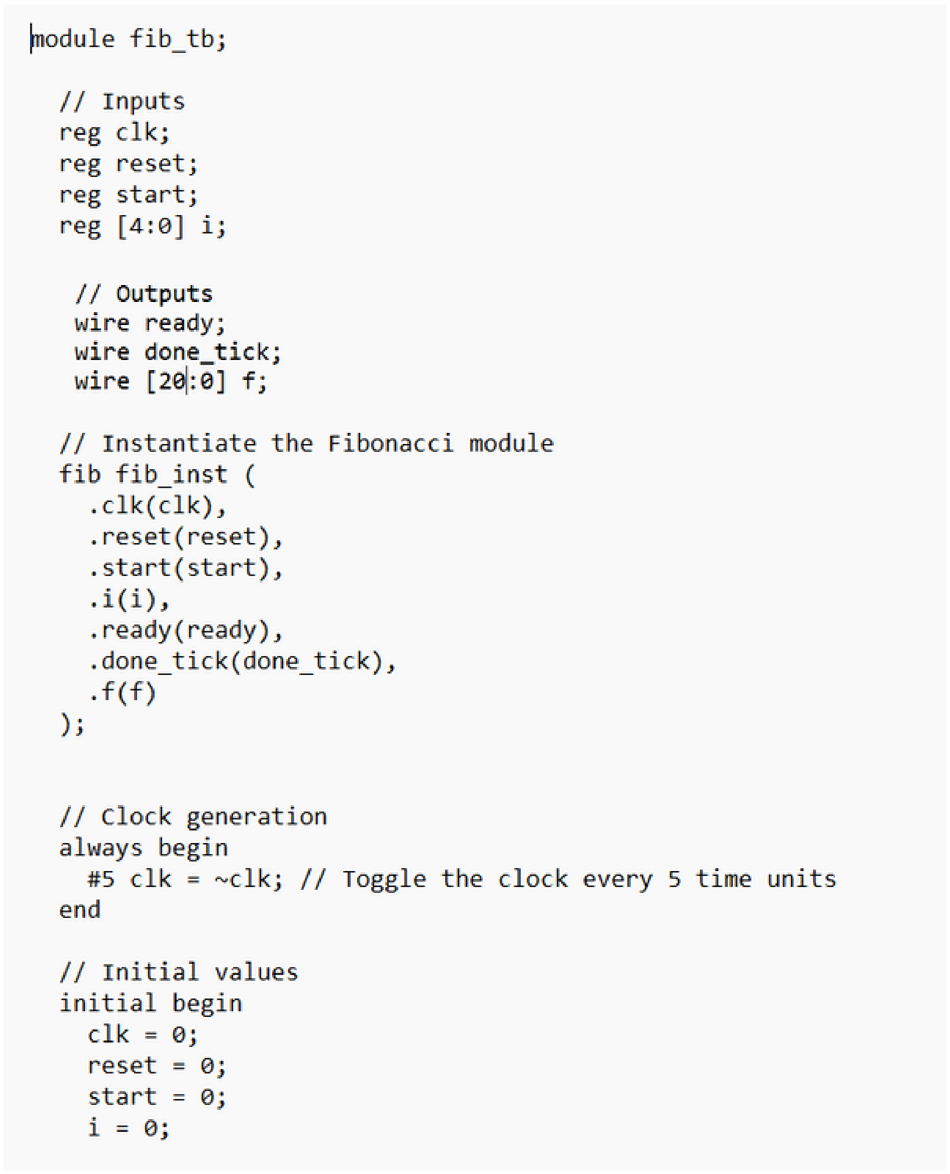
3.1 Overview of verification methodologies used

The verification process relies on a multifaceted approach, incorporating formal methods, simulation-based testing, and code coverage analysis. This amalgamation ensures a comprehensive assessment of the design's correctness, covering a spectrum of potential issues.

3.2 Explanation of normal testbench and code coverage

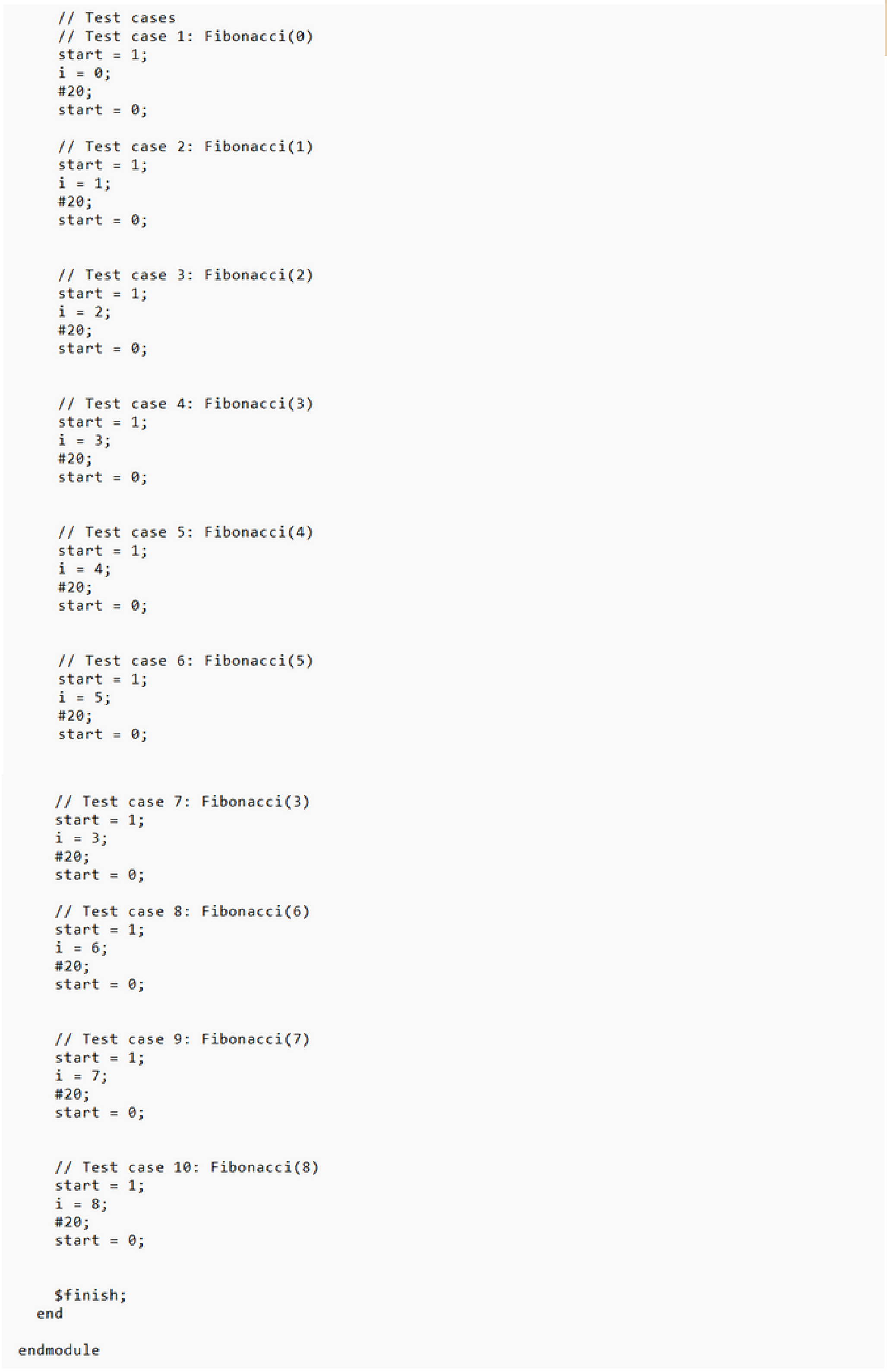
The normal testbench constitutes a dynamic simulation, subjecting the design to diverse scenarios to validate its functionality. Concurrently, code coverage analysis meticulously gauges the extent to which the design's code is exercised during simulation, providing invaluable insights into areas that remain untested.

Normal testbench for Fibonacci series



```
module fib_tb;

  // Inputs
  reg clk;
  reg reset;
  reg start;
  reg [4:0] i;

   // Outputs
   wire ready;
   wire done_tick;
   wire [20:0] f;

  // Instantiate the Fibonacci module
  fib fib_inst (
    .clk(clk),
    .reset(reset),
    .start(start),
    .i(i),
    .ready(ready),
    .done_tick(done_tick),
    .f(f)
  );


  // Clock generation
  always begin
    #5 clk = ~clk; // Toggle the clock every 5 time units
  end

  // Initial values
  initial begin
    clk = 0;
    reset = 0;
    start = 0;
    i = 0;
```

Normal Testbench



```
    // Test cases
    // Test case 1: Fibonacci(0)
    start = 1;
    i = 0;
    #20;
    start = 0;

    // Test case 2: Fibonacci(1)
    start = 1;
    i = 1;
    #20;
    start = 0;


    // Test case 3: Fibonacci(2)
    start = 1;
    i = 2;
    #20;
    start = 0;


    // Test case 4: Fibonacci(3)
    start = 1;
    i = 3;
    #20;
    start = 0;


    // Test case 5: Fibonacci(4)
    start = 1;
    i = 4;
    #20;
    start = 0;


    // Test case 6: Fibonacci(5)
    start = 1;
    i = 5;
    #20;
    start = 0;



    // Test case 7: Fibonacci(3)
    start = 1;
    i = 3;
    #20;
    start = 0;

    // Test case 8: Fibonacci(6)
    start = 1;
    i = 6;
    #20;
    start = 0;


    // Test case 9: Fibonacci(7)
    start = 1;
    i = 7;
    #20;
    start = 0;


    // Test case 10: Fibonacci(8)
    start = 1;
    i = 8;
    #20;
    start = 0;


    $finish;
  end

endmodule
```

Normal Testbench

Instantiate the Fibonacci module
This section instantiates the fib module and connects its input and output ports to signals declared in the testbench. This allows the testbench to control the inputs and monitor the outputs of the fib module.

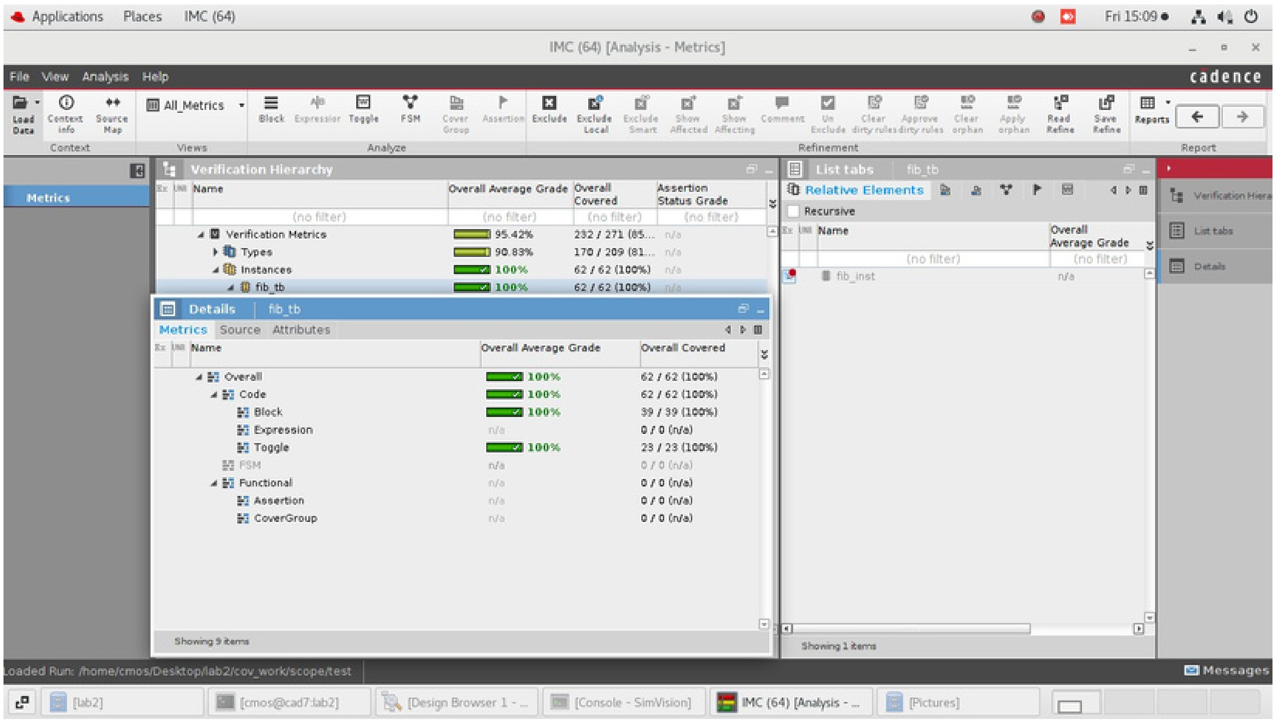
Clock generation
This part of the testbench generates a clock signal (**clk**) by toggling its value every 5 time units. This clock is used to drive the **fib** module.

Initial values

- Initializes the input signals (clk, reset, start, and i) to default values.
- Performs a reset sequence to ensure the fib module starts from a known state.
- Defines several test cases where different values of i are used to test various iterations of the Fibonacci computation.

Test cases

- This section defines individual test cases by setting values for start and i.
- For each test case, it asserts start to initiate the computation and sets the desired value for i.
- After a delay of 20 time units (#20), it deasserts start to end the computation.



IMC (64) [Analysis - Metrics]

Verification Hierarchy

Name	Overall Average Grade	Overall Covered	Assertion Status Grade
Verification Metrics	95.42%	232 / 271 (85...	n/a
Types	90.83%	170 / 209 (81...	n/a
Instances	100%	62 / 62 (100%)	n/a
fib_tb	100%	62 / 62 (100%)	n/a

Details fib_tb — Metrics

Name	Overall Average Grade	Overall Covered
Overall	100%	62 / 62 (100%)
Code	100%	62 / 62 (100%)
Block	100%	39 / 39 (100%)
Expression	n/a	0 / 0 (n/a)
Toggle	100%	23 / 23 (100%)
FSM	n/a	0 / 0 (n/a)
Functional	n/a	0 / 0 (n/a)
Assertion	n/a	0 / 0 (n/a)
CoverGroup	n/a	0 / 0 (n/a)

Showing 9 items

List tabs fib_tb — Relative Elements

Name	Overall Average Grade
fib_inst	n/a

Showing 1 items

Loaded Run: /home/cmos/Desktop/lab2/cov_work/scope/test

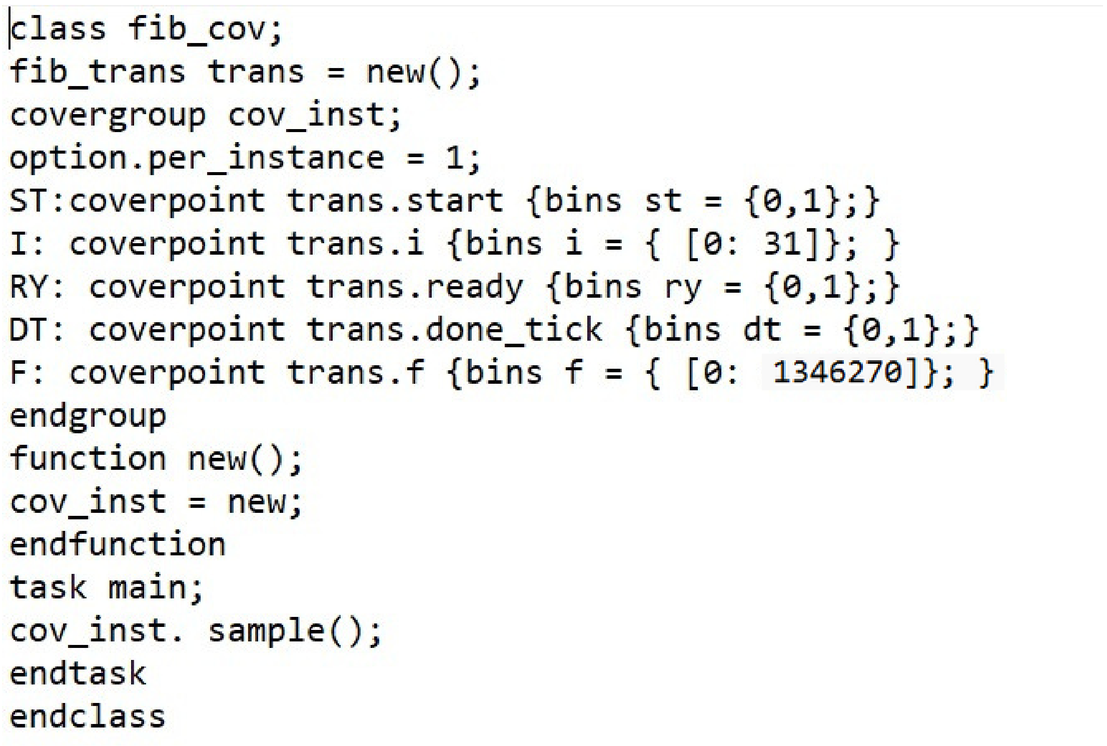
CODE COVERAGE FOR NORMAL TESTBENCH

The verification process relies on a multifaceted approach, incorporating formal methods, simulation-based testing, and code coverage analysis. This amalgamation ensures a comprehensive assessment of the design's correctness, covering a spectrum of potential issues.

3.3 Layered testbench structure

A strategic layered testbench structure is implemented to scrutinize individual modules before their integration into the complete system. This methodical approach ensures that each component functions with precision before amalgamation, thus fortifying the overall integrity of the design.

fib_cov



```
class fib_cov;
fib_trans trans = new();
covergroup cov_inst;
option.per_instance = 1;
ST:coverpoint trans.start {bins st = {0,1};}
I: coverpoint trans.i {bins i = { [0: 31]}; }
RY: coverpoint trans.ready {bins ry = {0,1};}
DT: coverpoint trans.done_tick {bins dt = {0,1};}
F: coverpoint trans.f {bins f = { [0:  1346270]}; }
endgroup
function new();
cov_inst = new;
endfunction
task main;
cov_inst. sample();
endtask
endclass
```

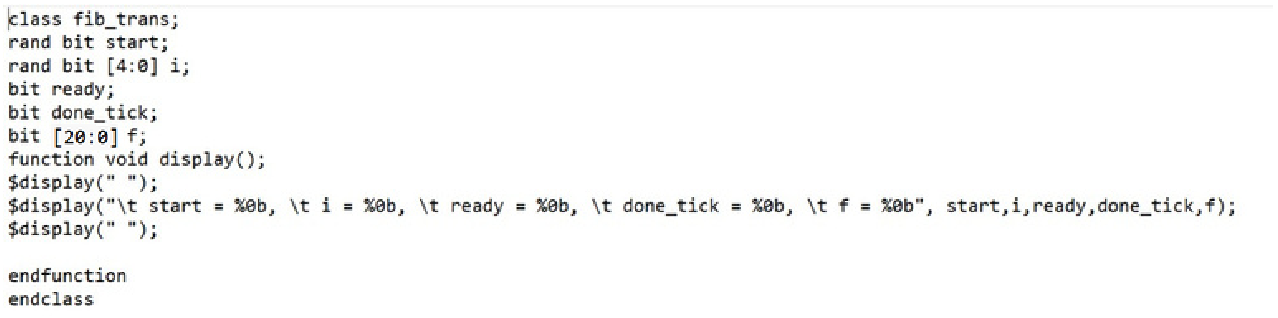
fib_intf

```
interface fib_intf(input logic clk,reset);
logic start;
logic [4:0] i;

logic  ready;
logic done_tick;
logic [20:0] f;

clocking bfm_cb @(posedge clk);
default input #1 output #1;
output start;
output i;
input ready;
input done_tick;
input f;
endclocking

clocking monitor_cb @(posedge clk);
default input #1 output #1;
input start;
input i;
input ready;
input done_tick;
input f;
endclocking
modport BFM (clocking bfm_cb, input clk,reset);
modport MONITOR (clocking monitor_cb, input clk, reset);
endinterface
```

fib_bfm

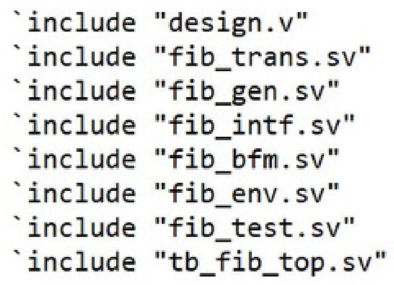
```
class fib_bfm;
virtual fib_intf intf;
mailbox gen2bfm;
int no_transactions;
function new(virtual fib_intf intf,mailbox gen2bfm);
this.intf = intf;
this.gen2bfm = gen2bfm;
endfunction
task reset;

wait(intf.reset);
$display("Reset Initiated");
intf.bfm_cb.start <= 0;
intf.bfm_cb.i <= 0;
wait(!intf.reset);
$display("Reset finished");
endtask



task main;
forever begin
fib_trans trans;
gen2bfm.get(trans);
$display("Transaction No. = %0d", no_transactions);
intf.bfm_cb.start <= trans. start;
intf.bfm_cb.i <= trans.i;
repeat(2)@(posedge intf.clk);
trans.ready = intf.bfm_cb.ready;
trans.done_tick = intf.bfm_cb.done_tick;
trans.f = intf.bfm_cb.f;
trans.display();
no_transactions++;
end
endtask
endclass
```

fib_trans



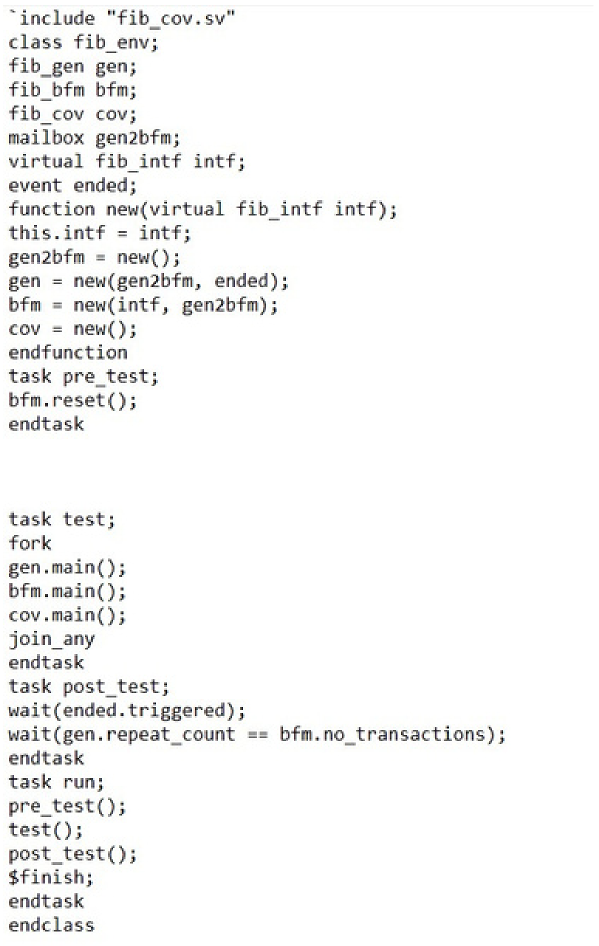
```
class fib_trans;
rand bit start;
rand bit [4:0] i;
bit ready;
bit done_tick;
bit [20:0] f;
function void display();
$display(" ");
$display("\t start = %0b, \t i = %0b, \t ready = %0b, \t done_tick = %0b, \t f = %0b", start,i,ready,done_tick,f);
$display(" ");

endfunction
endclass
```

defines



```
`include "design.v"
`include "fib_trans.sv"
`include "fib_gen.sv"
`include "fib_intf.sv"
`include "fib_bfm.sv"
`include "fib_env.sv"
`include "fib_test.sv"
`include "tb_fib_top.sv"
```

fib_env



```
`include "fib_cov.sv"
class fib_env;
fib_gen gen;
fib_bfm bfm;
fib_cov cov;
mailbox gen2bfm;
virtual fib_intf intf;
event ended;
function new(virtual fib_intf intf);
this.intf = intf;
gen2bfm = new();
gen = new(gen2bfm, ended);
bfm = new(intf, gen2bfm);
cov = new();
endfunction
task pre_test;
bfm.reset();
endtask


task test;
fork
gen.main();
bfm.main();
cov.main();
join_any
endtask
task post_test;
wait(ended.triggered);
wait(gen.repeat_count == bfm.no_transactions);
endtask
task run;
pre_test();
test();
post_test();
$finish;
endtask
endclass
```

fib_test

```
program fib_test(fib_intf intf);
fib_env env;
initial begin
env = new(intf);
env.gen.repeat_count = 200;
env.run();
end
endprogram
```

fib_gen

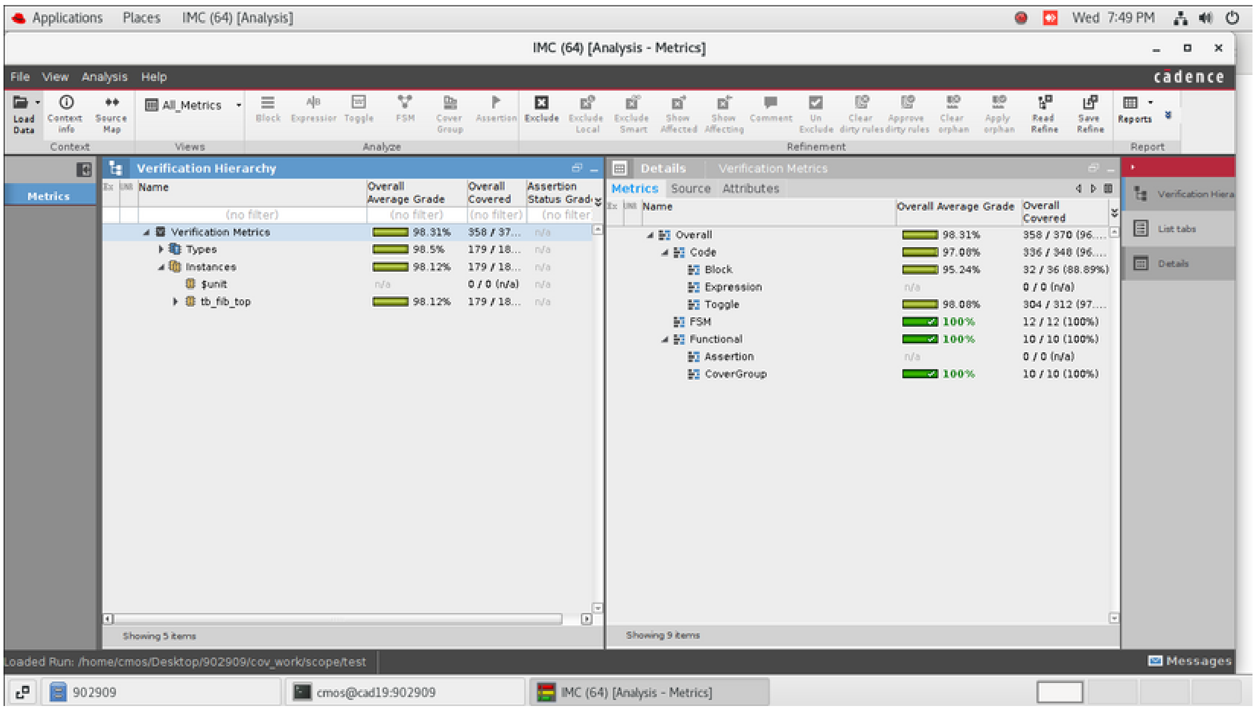
```
class fib_gen;
rand fib_trans trans;
mailbox gen2bfm;
event ended;
int repeat_count;
function new(mailbox gen2bfm, event ended);
this.gen2bfm = gen2bfm;
this.ended = ended;
endfunction
task main;
repeat(repeat_count) begin
trans = new();
if(!trans.randomize()) $display("Randomization Failed");
gen2bfm.put(trans);
end
->ended;
endtask
endclass
```

tb_fib_top

```
module tb_fib_top;
bit clk;
bit reset;
fib_intf intf(clk,reset);
fib_test test(intf);
fib dut(.clk(intf.clk),
.reset(intf.reset),
. start (intf.start),
.i(intf.i),
.ready(intf.ready),
.done_tick(intf.done_tick),
.f(intf.f)
);
always #5 clk = ~clk;
initial begin
reset = 1;
#5;
reset = 0;
end
endmodule
```

interface

```
interface fib_intf(input logic clk,reset);
logic start;
logic [4:0] i;

logic  ready;
logic done_tick;
logic [20:0] f;

clocking bfm_cb @(posedge clk);
default input #1 output #1;
output start;
output i;
input ready;
input done_tick;
input f;
endclocking

clocking monitor_cb @(posedge clk);
default input #1 output #1;
input start;
input i;
input ready;
input done_tick;
input f;
endclocking
modport BFM (clocking bfm_cb, input clk,reset);
modport MONITOR (clocking monitor_cb, input clk, reset);
endinterface
```


Verification Hierarchy

Name	Overall Average Grade	Overall Covered	Assertion Status Grade
Verification Metrics	98.31%	358 / 37...	n/a
Types	98.5%	179 / 18...	n/a
Instances	98.12%	179 / 18...	n/a
$unit	n/a	0 / 0 (n/a)	n/a
tb_fib_top	98.12%	179 / 18...	n/a

Showing 5 items

Details — Verification Metrics

Name	Overall Average Grade	Overall Covered
Overall	98.31%	358 / 370 (96....
Code	97.08%	336 / 348 (96....
Block	95.24%	32 / 36 (88.89%)
Expression	n/a	0 / 0 (n/a)
Toggle	98.08%	304 / 312 (97....
FSM	100%	12 / 12 (100%)
Functional	100%	10 / 10 (100%)
Assertion	n/a	0 / 0 (n/a)
CoverGroup	100%	10 / 10 (100%)

Showing 9 items

Loaded Run: /home/cmos/Desktop/902909/cov_work/scope/test

CODE COVERAGE FOR LAYERED TESTBENCH

4. Testbench Implementations

4.1 Simulation results and analysis

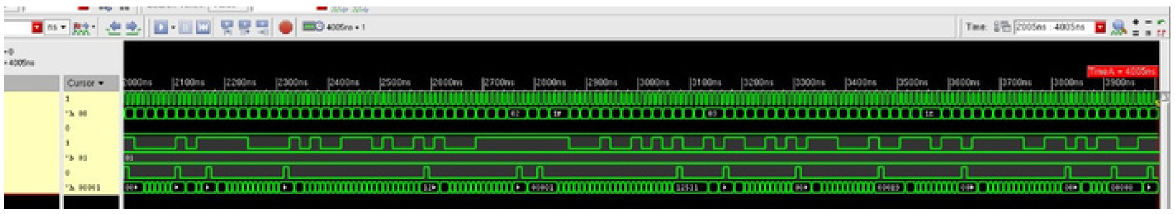


Fig 4.1 simulation of layered testbench

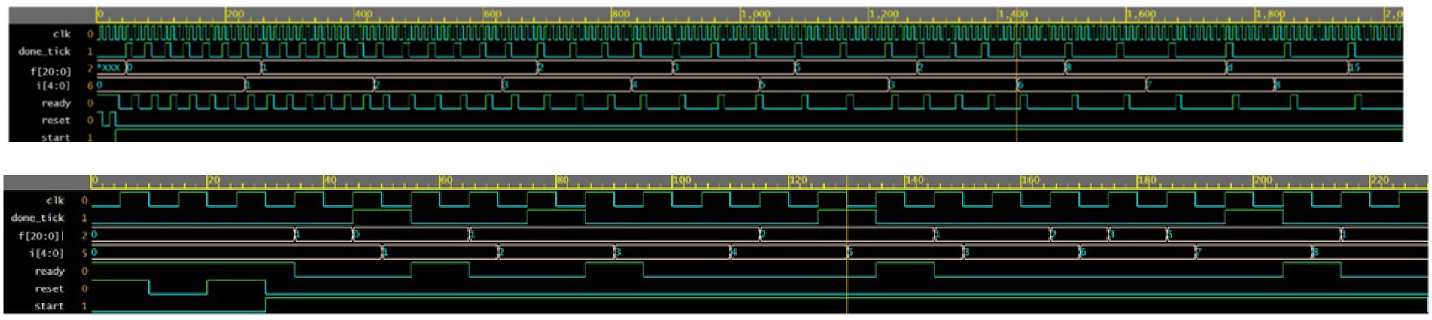


Fig 4.2 zoomed output of layered testbench (a,b)

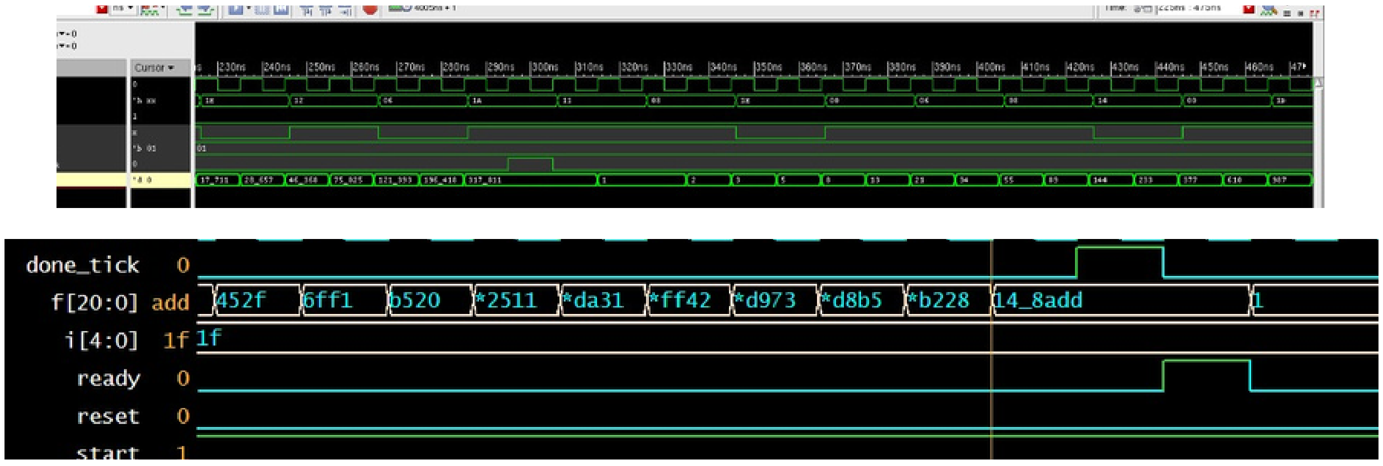
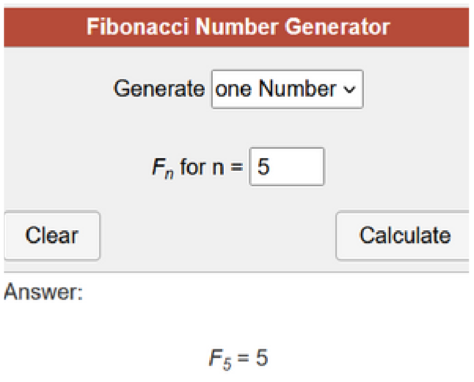


Fig 4.3 zoomed output of layered testbench with done tick going high (a, b)

Simulation result

Fig4.1 gives us the simulation waveform

Fig4.2 represents the waveform for the Fibonacci series, to confirm these values we have fig 4.4 where the input i=5 and hence the output f=5 and in fig 4.5 i=10 and f=55, these values correspond to out waveform hence confirming accuracy of the code.

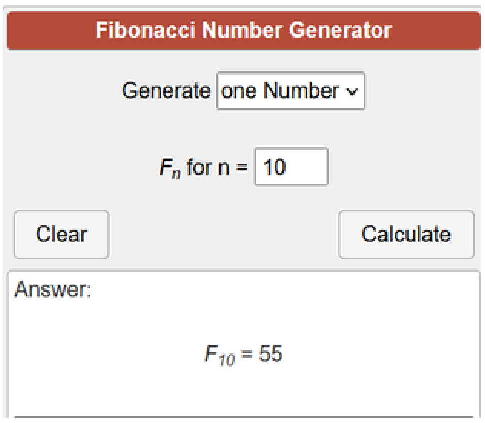


F_n for n = 5

Answer:

$F_5 = 5$

Fig 4.4 Fibonacci calclator with i=5 and f=5.



F_n for n = 10

Answer:

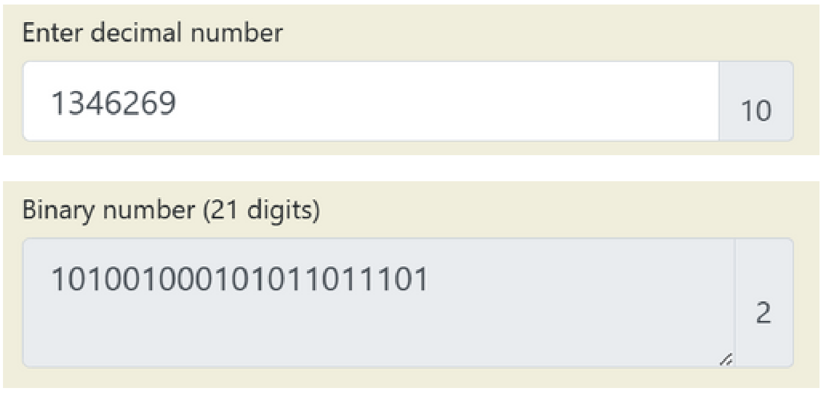
$F_{10} = 55$

Fig 4.4 Fibonacci calclator with i=10 and f=55.

Fig4.3 shows that when we take a high value for input, we get 0 as an output, this is because we have set an upper limit at i=28 and if i>28 the output for these signals done_tick high, meaning completion.

Simulation result

Fig4.3 b we have discovered that we get the output for the i = 32 correctly when we apply f with 21 bits


Enter decimal number

1346269 | 10

Binary number (21 digits)

101001000101011011101 | 2

fibonacci series

```
0 : 0
1 : 1
2 : 1
3 : 2
4 : 3
5 : 5
6 : 8 = 2^3
7 : 13
8 : 21 = 3 x 7
9 : 34 = 2 x 17
10 : 55 = 5 x 11
11 : 89
12 : 144 = 2^4 x 3^2
13 : 233
14 : 377 = 13 x 29
15 : 610 = 2 x 5 x 61
16 : 987 = 3 x 7 x 47
17 : 1597
18 : 2584 = 2^3 x 17 x 19
19 : 4181 = 37 x 113
20 : 6765 = 3 x 5 x 11 x 41
21 : 10946 = 2 x 13 x 421
22 : 17711 = 89 x 199
23 : 28657
24 : 46368 = 2^5 x 3^2 x 7 x 23
25 : 75025 = 5^2 x 3001
26 : 121393 = 233 x 521
27 : 196418 = 2 x 17 x 53 x 109
28 : 317811 = 3 x 13 x 29 x 281
29 : 514229
30 : 832040 = 2^3 x 5 x 11 x 31 x 61
31 : 1346269 = 557 x 2417
```

Analysis

Normal testbench alone is not effective enough in terms of testing and achieving sufficient code coverage.

On the other hand, the layered testbench, with its structured approach, seems to be more successful in reaching the desired level of code coverage.

4.2 Purpose and structure

The layered testbench helps to improve the overall dependability of each module by isolating and addressing problems at a specific level.
A layered testbench is a testing approach in which the testbench is divided into several layers, each with a designated function. This architectural approach promotes modularity, reusability, and maintainability in complex verification environments.

Structure

Fibonacci Module (fib):

- The actual hardware module under test.

BFM(Bus Functional Model) Layer (fib_bfm):

- Serves as an abstract representation of the communication between the testbench and the hardware module.
- Handles tasks related to resetting the module, processing transactions, and interfacing with the **fib_intf** interface.

Interface Layer (fib_intf):

- Defines the interface signals that facilitate communication between the testbench and the hardware module.
- Includes clocking blocks (**bfm_cb** and **monitor_cb**) for proper synchronization of signals.

Generator Layer (fib_gen):

- Responsible for generating random transactions (**fib_trans**) for stimulus.

Coverage Layer (fib_cov):

- Defines a coverage group (**cov you_inst**) to track and analyze the coverage of specific signals in the hardware module.
- Helps assess the effectiveness of the test suite in exercising different parts of the design.

Test Layer (fib_test):

- Orchestrates the overall test flow.
- Instantiates the environment (fib_env) and sets up the necessary parameters.
- Runs the simulation by calling the run task in the environment.

Environment Layer (fib_env):

- Integrates and coordinates various components of the testbench, including the generator, BFM, and coverage group.
- Manages the initialization, execution, and post-processing of the test.

Top-Level Testbench (tb_fib_top):

- Instantiates the fib_intf interface, the fib_test program, and the hardware module (fib).
- Controls the

Purpose

Modularity: Each layer is designed to perform a specific set of tasks, making it easier to understand, modify, and extend the testbench.
Reusability: Components like the generator, BFM, and coverage group can be reused in other projects or testbenches.
Maintainability: Changes or enhancements to one layer can be made independently without affecting other layers, making the testbench easier to maintain.
Scalability: As the complexity of the design increases, a layered testbench can be extended by adding more layers or refining existing ones.
Coverage Analysis: The coverage layer provides insights into how well the design has been exercised during simulation, helping identify areas that need additional testing.

5.Coverage Analysis

5.1 Cover points definition and relevance to the design

Cover points, strategically defined, track specific events or conditions during simulation. These points are intricately selected based on their relevance to critical functionalities within the design, serving as benchmarks for the assessment of testing thoroughness. The cover points within the covergroup cov_inst are associated with specific signals from the fib_trans class (trans).

A layered testbench provides a structured and organized approach to code coverage, offering benefits in terms of modularity, reusability, hierarchical analysis, scalability, and ease of debugging.

5.2 Code coverage metrics and analysis

Code coverage metrics, encompassing statement coverage, branch coverage, and condition coverage, are meticulously employed to quantitatively measure the effectiveness of the testbenches in exercising the entirety of the design code.

6. Conclusion

Throughout the project, we successfully designed the Fibonacci series code, implemented comprehensive testbenches, and obtained accurate results using Cadence. In addition, the utilization of layered testbenches proved instrumental, showcasing enhanced code coverage compared to traditional test cases.

Advantages of Layered Testbenches:

1. Modularity:
    - Inherently modular, focusing on specific verification aspects.
    - Facilitates easier understanding, maintenance, and extension.
2. Reusability:
    - Enables reuse across various projects or within the same project.
    - Promotes time savings and a standardized approach.
3. Scalability:
    - Adapts to design complexity by adding new layers.
    - Beneficial for managing large and intricate designs.
4. Ease of Debugging:
    - Simplifies debugging by isolating issues to specific layers.
    - Streamlines identification and resolution of problems.

5. Abstraction:
    - Offers a higher level of abstraction, enhancing readability.
    - Adaptable to changes in design or verification requirements.

In summary, layered testbenches not only improved code coverage but also provided modularity, reusability, scalability, ease of debugging, and abstraction, boosting the efficiency of the verification process.