Assignment No.4

EE5530 : Principles of SoC Functional Verification

Nakul C - 122101024

Problem Statement

This assignment involves modifying an existing SystemVerilog testbench for a GCD module by introducing a modular architecture. The enhanced testbench will include components such as a generator, driver, monitor, scoreboard, interface, clocking block, and synchronization using mailboxes and events. These will be organized within an environment and controlled through a test module. The complete setup will be simulated and validated on EDA Playground.

Device Under Test (DUT): GCD Module

The Device Under Test (DUT) for this project is a hardware implementation of the **Greatest Common Divisor (GCD)** computation module. The DUT follows a modular design, split into two major subcomponents: the **datapath** and the **control path**, both of which are instantiated and interconnected within the `top_module`.

Overview

The purpose of the DUT is to compute the GCD of two 8-bit input numbers, **A_in** and **B_in**. It implements an iterative version of the Euclidean algorithm, continuously subtracting the smaller number from the larger until one of them becomes zero. The remaining non-zero number is the GCD.

The DUT also includes control and handshake signals to interact with a testbench or a driver module, ensuring synchronization and correctness of operation.

Top Module

The `top_module` instantiates and connects the datapath and control path modules. It manages the input and output signals and routes them appropriately.

- **Inputs:**
  - `A_in` (8-bit): First operand for the GCD computation.
  - `B_in` (8-bit): Second operand for the GCD computation.
  - `operands_val`: Indicates that the input operands are valid and the computation can begin.
  - `Clk`: Clock signal for synchronous operations.

- `Rst`: Asynchronous reset signal to initialize the system.
- `ack`: Acknowledgement signal used to reset the DUT after computation is complete.

- **Outputs:**
  - `gcd_out` (8-bit): Output representing the computed GCD.
  - `gcd_valid`: Flag indicating that the output `gcd_out` is valid and ready to be read.

Datapath Module

The `datapath` module performs the core arithmetic operations required by the GCD algorithm. It consists of the following components:

- Two 8-bit registers **A** and **B** to store the operands.
- Multiplexers to select between inputs and outputs for updating the registers.
- Combinational logic for subtraction and comparisons (`A_lt_B`, `B_eq_0`).

The datapath is controlled by enable signals and select lines generated by the control path.

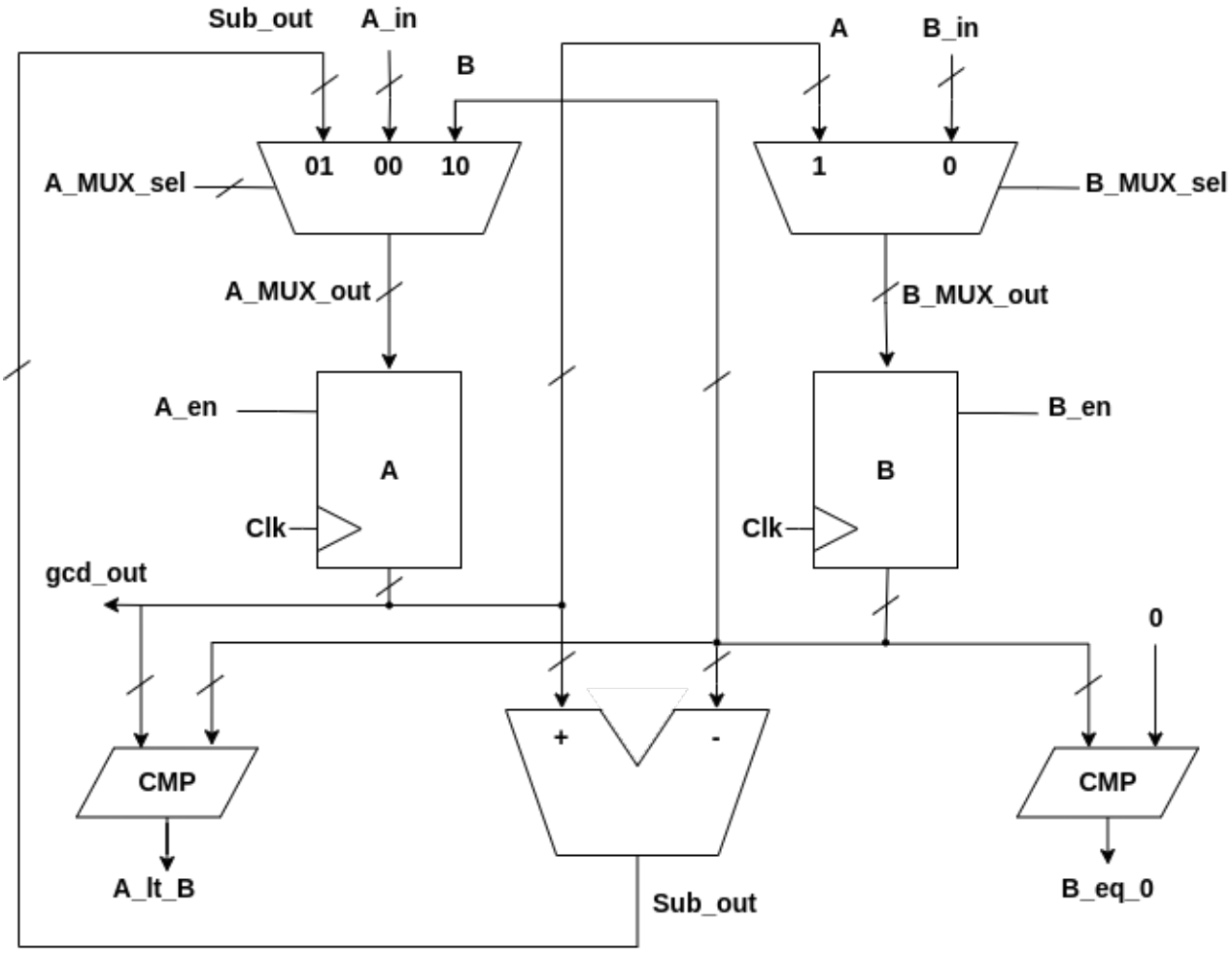

Figure 1: Architecture for the data path

Control Path Module

The `controlpath` module is a finite state machine (FSM) responsible for controlling the datapath operations. It has three main states:

1. **idle**: Waits for valid operands.
2. **busy**: Executes the iterative subtraction steps of the GCD algorithm.
3. **done**: Signals the end of computation and waits for an acknowledgment to reset.

The control path generates:

- Multiplexer select signals for A and B inputs.
- Enable signals for updating registers.
- `gcd_valid` signal indicating computation completion.

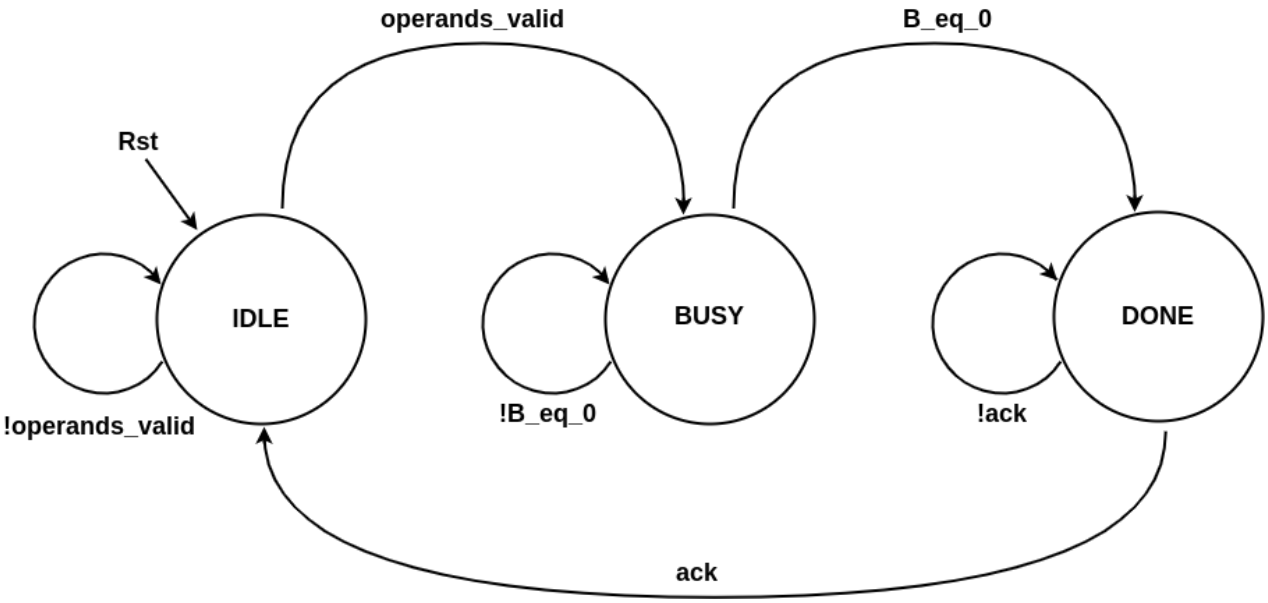

Figure 2: FSM for the states of the system

Testbench Architecture

The testbench is structured to verify the functionality of the **GCD module** using a modular approach. It includes several SystemVerilog classes and interfaces organized to simulate and validate the **design under test (DUT)**. Below is an overview of the components used:

- **Interface** (`gcd_if`) – Connects the DUT and testbench components, carrying signals such as inputs, outputs, clock, and reset.
- **Package** (`gcd_pkg`) – Encapsulates all class definitions, including the packet, **generator**, **driver**, **monitor**, **scoreboard**, and DUT model.
- **Generator** – Produces randomized input transactions and stores the expected results in a queue.

- **Driver** – Drives input values from the generator to the DUT through the interface.
- **Monitor** – Observes outputs from the DUT and forwards them to the scoreboard.
- **Scoreboard** – Compares DUT outputs with expected results to validate correctness.
- **Mailboxes** – Used for communication between the generator and driver.
- **Clocking** – A 100MHz clock is generated using a simple `always` block.
- **Simulation Flow** – The generator is run first to populate transactions. Then, the driver and monitor run concurrently, and the scoreboard evaluates the outputs.

The simulation is controlled from a **top-level module** (`tb_top`) where components are instantiated, connected, and executed. The simulation logs key events and terminates upon completion of all transactions.

GCD Interface (`gcd_if`)

The **gcd_if** is a SystemVerilog interface that groups all the signals used to interact with the GCD (Greatest Common Divisor) hardware module. It includes the input signals **A_in** and **B_in** for supplying the two numbers whose GCD needs to be calculated. The **operands_val** signal indicates when valid inputs are being sent, and **ack** is used to acknowledge receipt of the GCD result. The **Rst** signal resets the DUT. On the output side, **gcd_out** provides the calculated GCD, and **gcd_valid** indicates when the result is ready.

The interface also includes a **clocking block** named **cb**, which synchronizes signal interaction with the rising edge of the clock **Clk**. The clocking block defines which signals are driven by the testbench (**output**) and which ones are read from the DUT (**input**), helping maintain proper timing and avoiding race conditions. Using an interface like this makes the testbench cleaner and keeps the communication with the DUT well-organized.

```
1  // GCD Interface
2  interface gcd_if(input bit Clk);
3      logic [7:0] A_in;
4      logic [7:0] B_in;
5      logic operands_val;
6      logic ack;
7      logic [7:0] gcd_out;
8      logic gcd_valid;
9      logic Rst;
10
11     // Clocking blocks
12     clocking cb @(posedge Clk);
13         output A_in, B_in, operands_val, ack, Rst;
14         input gcd_out, gcd_valid;
15     endclocking
16 endinterface
```

Transaction Class (`gcd_packet`)

The **gcd_packet** class represents a transaction object that holds the data required to test the GCD (Greatest Common Divisor) hardware module. It includes two randomized 8-bit

operands, **A** and **B**, which are the inputs to the GCD calculation. The **operands_valid** flag indicates whether the operands are valid and ready to be sent to the DUT (Device Under Test). The constructor function **new()** initializes the **operands_valid** signal to logic high (**1'b1**) by default.

A constraint block named **c_valid** ensures that the randomized values of **A** and **B** fall within a meaningful range of 10 to 200. This avoids trivial cases (like zeros or ones) that could lead to less meaningful test scenarios, ensuring more robust testing.

The **display()** function is provided for debugging purposes. It prints out the values of **A**, **B**, and **operands_valid** with an optional prefix string to identify the source of the message.

Additionally, the class includes a **copy()** function that creates a deep copy of the current transaction object. This ensures that when a packet is added to a queue, it retains its values independently, avoiding unintended modifications due to reference sharing. This design makes the testbench more modular, readable, and easier to debug.

```
// Transaction Class
class gcd_packet;
    rand bit [7:0] A;
    rand bit [7:0] B;
    bit operands_valid;

    function new();
        operands_valid = 1'b1;
    endfunction

    // Constraint to make A and B meaningful
    constraint c_valid {
        A inside {[10:200]};  // Avoid trivial cases
        B inside {[10:200]};
    }

    // Displaying the generated packets for debugging
    function void display(string prefix = "");
        $display("%s A = %0d, B = %0d, operands_valid = %0b", prefix, A,
            B, operands_valid);
    endfunction

    // Deep copy method
    function gcd_packet copy();
        gcd_packet pkt_copy = new();
        pkt_copy.A = this.A;
        pkt_copy.B = this.B;
        pkt_copy.operands_valid = this.operands_valid;
        return pkt_copy;
    endfunction

endclass
```

Golden Reference Model (`calc_gcd()`)

The **calc_gcd** function is a SystemVerilog implementation of the golden reference model used to compute the Greatest Common Divisor (GCD) of two input integers, **a** and **b**. It uses the standard Euclidean algorithm to perform the calculation. The function initializes a temporary variable **temp** and enters a **while** loop that continues as long as **b** is not zero. Inside the loop, **temp** temporarily stores the value of **b**. The variable **b** is then updated with the remainder of **a** divided by **b** (i.e., **a % b**), and **a** is assigned the value of **temp**. An additional check is made: if **a** becomes zero, the function returns **b**. Once **b** becomes zero, the function exits the loop and returns **a**, which holds the final GCD result. This golden model serves as a reliable reference to verify the outputs of the DUT.

```
// GCD Function (Golden Model)
function int calc_gcd(input int a, input int b);
    int temp;

    // Special case: if a is 0, return 1

    while (b != 0) begin
        temp = b;
        b = a % b;
        a = temp;
        if (a == 0)
            return b;
    end

    return a;

endfunction
```

Driver Class

The **driver class** is a key component of the testbench responsible for driving the input values into the **Design Under Test (DUT)**. It interacts with the **generator** through a mailbox, retrieving packets containing input values and sending them to the DUT via the `gcd_if` interface. The class is designed to work in a continuous loop, consuming data from the mailbox, waiting for random delays to simulate real-world conditions, and then sending the data to the DUT. This helps in ensuring the DUT is tested under various input scenarios.

The `new()` function in the driver class initializes the driver by taking in the **interface** and the **mailbox** as arguments. The `vif` is a virtual interface handle that connects the driver to the DUT, while `gen2drv` is the mailbox used for receiving the input packets from the generator. This setup enables easy communication between components of the testbench while maintaining modularity. The function prepares the driver class to operate in the simulation by setting up these essential elements.

The `run()` task is the core of the **driver** functionality. It begins by asserting a reset pulse to the DUT, ensuring that the system starts from a known state. After the reset, the task enters a continuous loop where it checks if the mailbox contains any data from the generator. If the mailbox is empty, the driver will display a message and terminate. If packets are available, the driver retrieves them, waits for a random period (using `$urandom_range`), and then drives

the values of `A_in` and `B_in` to the DUT.

The task waits for the DUT to assert the `gcd_valid` signal, indicating that the computation has been completed, and the output is valid. After that, the driver asserts an acknowledgment signal `ack` to indicate that the DUT can reset or handle subsequent computations. This process continues indefinitely until the mailbox is empty or the task is completed. The random delays and acknowledgment mechanism mimic real-world processing times and provide a realistic simulation of the GCD computation sequence.

```
1
2  // Driver Class
3  class driver;
4      virtual gcd_if vif;
5      mailbox gen2drv;
6
7      function new(virtual gcd_if vif, mailbox gen2drv);
8          this.vif = vif;
9          this.gen2drv = gen2drv;
10     endfunction
11
12
13     task display_mailbox(mailbox mbox, string header = "[DRIVER] Mailbox
           Contents:");
14         automatic int count = mbox.num();
15         automatic gcd_packet pkt;
16         automatic gcd_packet queue_copy[$];  // Temporary queue to store
               items
17
18         $display("%s (Size: %0d)", header, count);
19
20         // Dequeue all items temporarily and display them
21         for (int i = 0; i < count; i++) begin
22             mbox.get(pkt);
23             queue_copy.push_back(pkt);
24
25             $display("  Packet[%0d] -> A = %0d, B = %0d",
26                      i, pkt.A, pkt.B);
27         end
28
29         // Put them back into the mailbox
30         foreach (queue_copy[i]) begin
31             mbox.put(queue_copy[i]);
32         end
33     endtask
34
35
36     task run();
37         gcd_packet pkt;
38
39         vif.cb.Rst <= 1;
40         repeat (2) @(vif.cb);  // Reset pulse
41         vif.cb.Rst <= 0;
42
```

```
43          forever begin
44
45            if (!gen2drv.num()) begin
46
47              $display("[%0t] DRIVER: Input mailbox is empty, finishing...
                    ", $time);
48              repeat (5) @(vif.cb);
49              break;
50
51                  end
52
53              // Get packet from generator
54              gen2drv.get(pkt);
55              // display_mailbox(gen2drv); // Used for debugging
56
57              // Wait random time before sending operands
58              repeat ($urandom_range(1, 5)) @(vif.cb);
59
60              vif.cb.A_in <= pkt.A;
61              vif.cb.B_in <= pkt.B;
62              vif.cb.operands_val <= pkt.operands_valid;
63
64             @(vif.cb);  // One clock
65
66              vif.cb.operands_val <= 0;  // De-assert after sending
67             @(vif.cb);
68
69              wait (vif.gcd_valid === 1);
70
71              repeat ($urandom_range(1, 5)) @(vif.cb);
72              vif.cb.ack <= 1;
73             @(vif.cb);
74              vif.cb.ack <= 0;
75
76            //$display("[%0t] DRIVER: Packet with A = %0d, B = %0d sent",
                  $time, pkt.A, pkt.B);// Used for debugging
77
78
79          end
80
81          $display("[%0t] DRIVER: Task completed!", $time);
82      endtask
83
84  endclass
```

Monitor

The **monitor class** is a crucial component in the testbench that observes the output signals of the **Design Under Test (DUT)**. It is responsible for monitoring the `gcd_valid` signal and verifying that the output produced by the DUT matches the expected results. The monitor interacts with the **scoreboard** to ensure the correctness of the GCD computation by checking the validity of the output whenever the `gcd_valid` signal is asserted.

The `new()` function initializes the monitor class by taking in two arguments: the `vif` (virtual interface) and the **scoreboard** handle. The `vif` provides the monitor access to the signals of the DUT, while the `sb` scoreboard allows the monitor to compare the observed output with the expected results stored in the scoreboard. This setup ensures that the monitor can effectively validate the DUT's performance during the simulation.

The `run()` task is the main operational component of the monitor class. It continuously monitors the `gcd_valid` signal by waiting for a positive edge on the clock (`posedge vif.Clk`). Whenever `gcd_valid` is asserted (indicating that the DUT has completed a computation), the monitor checks the output value `gcd_out` and calls the `check_output()` method of the scoreboard to verify the result. This check is performed only when the `gcd_valid` signal transitions from 0 to 1, ensuring that the monitor accurately detects valid output events.

The task also monitors the **expected output queue** in the scoreboard. If the queue is empty, indicating that all transactions have been processed and checked, the monitor displays a message and terminates the task. To ensure synchronization, the monitor waits for a few additional clock cycles before finishing. This process guarantees that all outputs are checked before the monitor completes its execution, ensuring that the DUT behaves as expected throughout the simulation.

```
1  // Monitor Class
2  class monitor;
3      virtual gcd_if vif;
4      scoreboard sb;
5
6
7    function new(virtual gcd_if vif,scoreboard sb);
8          this.vif = vif;
9          this.sb = sb;
10     endfunction
11
12
13
14     task run();
15         bit prev_gcd_valid = 0;
16
17         forever begin
18             @(posedge vif.Clk);
19
20             if (vif.gcd_valid === 1 && prev_gcd_valid === 0) begin
21                 // Check the output when gcd_valid is asserted
22                 sb.check_output(vif.gcd_out, $time);
23             end
24
25            prev_gcd_valid = vif.gcd_valid;
26
27             // Exit when the expect_queue in the scoreboard is empty
28           if (sb.expect_q.empty()) begin
29                 $display("[%0t] MONITOR: All transactions checked,
                       finishing...", $time);
```

```
30                          repeat (5) @(posedge vif.Clk);  // Add some
                                clock cycles for synchronization
31                  break;
32              end
33          end
34
35          $display("[%0t] MONITOR: Task completed!", $time);
36      endtask
37  endclass
```

Scoreboard

The **scoreboard class** is an essential component of the testbench responsible for tracking and comparing the expected results with the actual output produced by the **Design Under Test (DUT)**. It uses a queue, `expect_q`, to store the expected GCD results, and provides functionality to check the correctness of the DUT's output during the simulation. The scoreboard works closely with the monitor to ensure the accuracy of the computations performed by the DUT.

The `new()` function initializes the scoreboard class by taking a reference to the expected result queue (`expect_q[$]`) as an argument. This allows the scoreboard to access the expected results and compare them with the actual output from the DUT. The reference to the expected queue ensures that the scoreboard is always synchronized with the current state of the expected results during the simulation.

The `display_expect_q()` task is used to display the contents of the expected result queue (`expect_q`) for debugging purposes. It shows the size of the queue and the values stored in it, allowing users to track the expected outputs. This task is useful for visualizing the expected GCD values during simulation but is typically commented out during normal operation to avoid unnecessary output.

The `check_output()` task is called by the monitor whenever a valid output is received from the DUT. It compares the actual output from the DUT with the expected value popped from the front of the `expect_q` queue. If the values match, a `PASS` message is displayed; otherwise, a `FAIL` message is shown. This task ensures that the DUT is functioning correctly by verifying that its outputs align with the expected results, providing essential feedback during simulation.

```
1   // Monitor Class
2   class monitor;
3       virtual gcd_if vif;
4       scoreboard sb;
5
6
7     function new(virtual gcd_if vif,scoreboard sb);
8           this.vif = vif;
9           this.sb = sb;
10      endfunction
11
12
13
```

```
14      task run();
15          bit prev_gcd_valid = 0;
16
17          forever begin
18              @(posedge vif.Clk);
19
20              if (vif.gcd_valid === 1 && prev_gcd_valid === 0) begin
21                  // Check the output when gcd_valid is asserted
22                  sb.check_output(vif.gcd_out, $time);
23              end
24
25             prev_gcd_valid = vif.gcd_valid;
26
27              // Exit when the expect_queue in the scoreboard is empty
28            if (sb.expect_q.empty()) begin
29                  $display("[%0t] MONITOR: All transactions checked,
                        finishing...", $time);
30                          repeat (5) @(posedge vif.Clk);  // Add some
                              clock cycles for synchronization
31                  break;
32              end
33          end
34
35          $display("[%0t] MONITOR: Task completed!", $time);
36      endtask
37  endclass
```

Generator

The **generator class** is responsible for creating random test data to validate the functionality of the **Design Under Test (DUT)**. It generates random input packets of GCD values, which are then sent to the **driver** for processing. This interaction is facilitated through a mailbox (`gen2drv`), allowing the generator to communicate with the driver while keeping both components decoupled for better modularity.

The `run()` task is the core function of the generator class. It runs a loop 10 times, generating a new `gcd_packet` on each iteration. Each packet is populated with random values using the `randomize()` function and displayed for debugging. The generated packet is then placed into the `gen2drv` mailbox, from where the driver retrieves it for further processing. Additionally, the expected GCD result for each packet is computed using the `calc_gcd()` function and stored in the `expect_q` queue for later comparison by the scoreboard.

By generating random input data and computing the expected results, the generator class ensures that the DUT is tested with diverse and unpredictable test cases. This aids in thoroughly validating the DUT's GCD functionality. The separation of packet generation, input delivery to the driver, and expected result storage in the scoreboard enables a well-structured and modular testbench.

```
1  // Generator Class
2
3  class generator;
4
```

```
5     mailbox gen2drv;
6     int expect_q[$];
7
8     function new(mailbox gen2drv);
9       this.gen2drv = gen2drv;
10    endfunction
11
12    task run();
13      repeat (10) begin
14        gcd_packet pkt = new();
15        pkt.randomize();
16        pkt.display("[GEN] Generated packet ->");
17
18        // Sending packet to driver
19        gen2drv.put(pkt);
20
21        // Saving expected output for scoreboard
22        expect_q.push_back(calc_gcd(pkt.A, pkt.B));
23      end
24    endtask
25
26  endclass
```

Package

The **gcd_pkg** package includes various components necessary for testing the GCD functionality of the Design Under Test (DUT). It imports the `gcd_packet`, `generator`, `calc_gcd`, `driver`, `scoreboard`, and `monitor` modules, which are responsible for packet generation, GCD calculation, driving values to the DUT, comparing outputs, and monitoring the simulation. These components work together to ensure the DUT is tested with a range of inputs and that its outputs are validated against the expected results.

```
1  package gcd_pkg;
2      `include "gcd_packet.sv"
3          `include "generator.sv"
4      `include "calc_gcd.sv"
5      `include "driver.sv"
6          `include "scoreboard.sv"
7      `include "monitor.sv"
8  endpackage
```

Testbench Top Module

The `tb_top` module serves as the top-level testbench for the GCD functionality simulation. It defines the main simulation setup, including the clock generation and connections to the Design Under Test (DUT). The clock is generated with a period of 10 ns, simulating a 100 MHz clock. The testbench uses the `gcd_if` interface to connect the DUT to the rest of the test components, allowing for the input and output of the operands and results to be driven and monitored during the simulation.

In the `initial` block, the testbench is initialized by setting the clock to zero and displaying

simulation start messages. The generator instance (`gen`) is then created and run to generate random GCD input packets, which are sent to the driver via a mailbox (`gen2drv`). This step ensures that the input data is randomized and ready for processing by the DUT. The generated packets are displayed for debugging purposes, helping to track the inputs as they are sent to the DUT.

Once the packets are generated, the expected GCD results are passed to the scoreboard, and the instances for the `scoreboard` (`sb`), `monitor` (`mon`), and `driver` (`drv`) are created. The driver takes the input packets from the mailbox and applies them to the DUT. Meanwhile, the monitor continuously checks the DUT's output and compares it against the expected results stored in the scoreboard. Both the driver and monitor run in parallel, allowing for simultaneous driving and checking of the DUT's behavior.

Finally, the testbench finishes by displaying simulation completion messages and invoking `$finish` to stop the simulation. This setup ensures that all test components — including the packet generator, driver, scoreboard, and monitor — work together to provide a comprehensive verification environment for the GCD functionality of the DUT. This modular structure allows for flexibility in testing and is an essential part of any SystemVerilog testbench.

```
1  `timescale 1ns/1ps
2
3  `include "gcd_if.sv"
4  `include "gcd_pkg.sv"
5  import gcd_pkg::*;
6
7
8  module tb_top;
9
10     bit Clk;
11     always #5 Clk = ~Clk;  // 100MHz clock
12     gcd_if gcd_vif(Clk);
13
14     // Mailboxes
15     mailbox gen2drv = new();
16
17     // Scoreboard expected values
18     int expect_q[$];
19
20     // DUT
21     top_module dut (
22         .A_in(gcd_vif.A_in),
23         .B_in(gcd_vif.B_in),
24         .operands_val(gcd_vif.operands_val),
25         .Clk(Clk),
26         .Rst(gcd_vif.Rst),
27         .ack(gcd_vif.ack),
28         .gcd_out(gcd_vif.gcd_out),
29         .gcd_valid(gcd_vif.gcd_valid)
30     );
31
32     // Class Handles
33     generator gen;
```

```
34      driver drv;
35      scoreboard sb;
36      monitor mon;
37
38      initial begin
39          Clk = 0;
40
41          $display("*********************************************");
42          $display("*************SIMULATION-STARTED*************");
43          $display("*********************************************");
44
45          // Creating instances
46          gen = new(gen2drv);
47
48          // Run generator first
49          fork
50              gen.run();
51          join
52
53          $display("*********************************************");
54          $display("*************PACKET-GENERATED***************");
55          $display("*********************************************");
56
57
58          // Send expected_q to scoreboard
59          expect_q = gen.expect_q;
60
61          // Creating instances
62          sb  = new(expect_q);
63          mon = new(gcd_vif, sb);
64          drv = new(gcd_vif, gen2drv);
65
66          // Run driver and monitor in parallel
67          fork
68              drv.run();
69              mon.run();
70          join
71
72          $display("*********************************************");
73          $display("*************SIMULATION-COMPLETE*************");
74          $display("*********************************************");
75
76          $finish;
77      end
78
79  endmodule
```

Simulation Log and Results

The simulation was executed using Synopsys VCS, as shown in the terminal output. The compilation and elaboration steps completed without any critical errors, although there were

warnings related to the use of queue methods that are extensions supported by VCS but not defined in the SystemVerilog LRM.

Packet Generation

A total of ten randomized test packets were generated. Each packet contains two operands, **A** and **B**, and an **operands_valid** flag. The generated packets are displayed in the simulation log:

```
[TB] Generated packet -> A = 15, B = 30, operands_valid = 1
[TB] Generated packet -> A = 42, B = 187, operands_valid = 1
...
[TB] Generated packet -> A = 144, B = 90, operands_valid = 1
```

Transaction Verification

For each packet, the expected GCD was calculated using the golden model function **calc_gcd()**. The DUT's output was compared against these expected results. The monitor displayed **PASS** messages confirming that the DUT's output matched the expected GCD for every transaction.

An example of the verification output is shown below:

```
[PASS] Time=95000 | DUT: 15 Expected: 15
[PASS] Time=365000 | DUT: 1 Expected: 1
[PASS] Time=625000 | DUT: 1 Expected: 1
[PASS] Time=865000 | DUT: 2 Expected: 2
```

Each message includes:

- Simulation time (in ps)
- DUT output value
- Expected output value from the golden model

Simulation Completion

After processing all transactions, both the driver and monitor components completed their tasks, and the simulation was successfully terminated. The final messages confirm that all transactions were verified and no mismatches were found:

```
[2455000] MONITOR: All transactions checked, finishing...
[2455000] DRIVER: Input queue empty, finishing...
[2505000] MONITOR: Task completed!
[2505000] DRIVER: Task completed!
********************************************************************************
****************************SIMULATION COMPLETED****************************
********************************************************************************
```

The simulation finished at time **2,505,000 ps** with all test cases passing successfully, verifying the correctness of the DUT for the generated input scenarios.

EDAPLAYGROUND Link

You can view or run the Verilog code in EDAPLAYGROUND

Click here to visit EDAPLAYGROUND