Assignment No.3

EE5530 : Principles of SoC Functional Verification

Nakul C - 122101024

Problem Statement

Enhance the GCD test bench to make it a class and queue-based test bench.

1. **Create a class gcd_packet which contains A, B, and an operands_valid. Create multiple packets of type gcd_packet by randomizing A and B to have meaningful values. Otherwise, the GCD will always be 1.**
2. **Push the packets into an input queue.**
3. **Populate an expect queue by calling a calc_gcd function which calculates the GCD of the given two numbers.**
4. **Create a run task/driver which:**
   - **Waits for a random amount of time**
   - **Pops the gcd_pkt from the input queue and sends it to the DUT**
   - **Sends the acknowledgement to the DUT after a random delay**
5. **Once gcd_valid is asserted, pop from the expect queue and compare with the DUT output.**
6. **Use clocking blocks and interfaces as well.**

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.

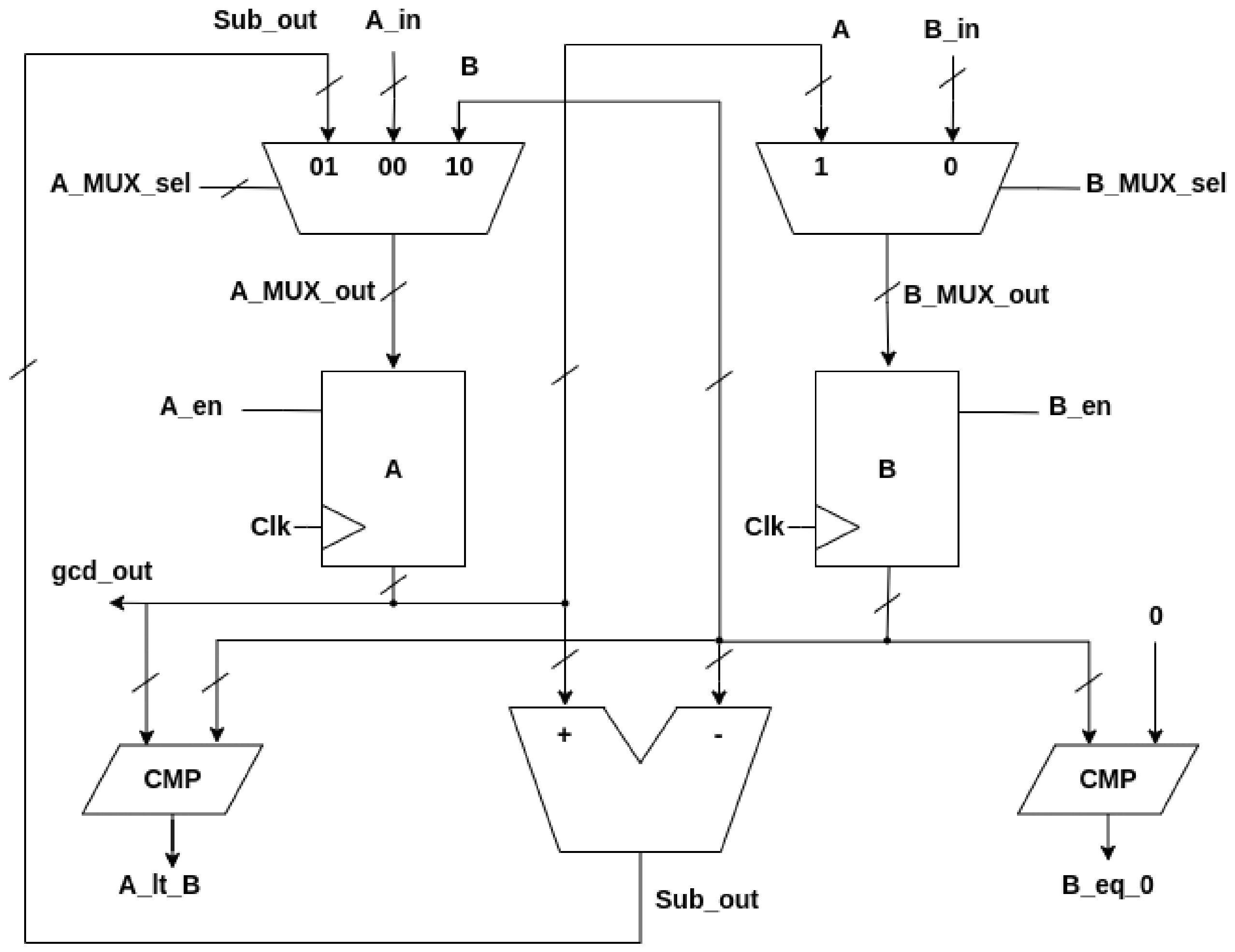

Figure 1: Architecture for the data path

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.

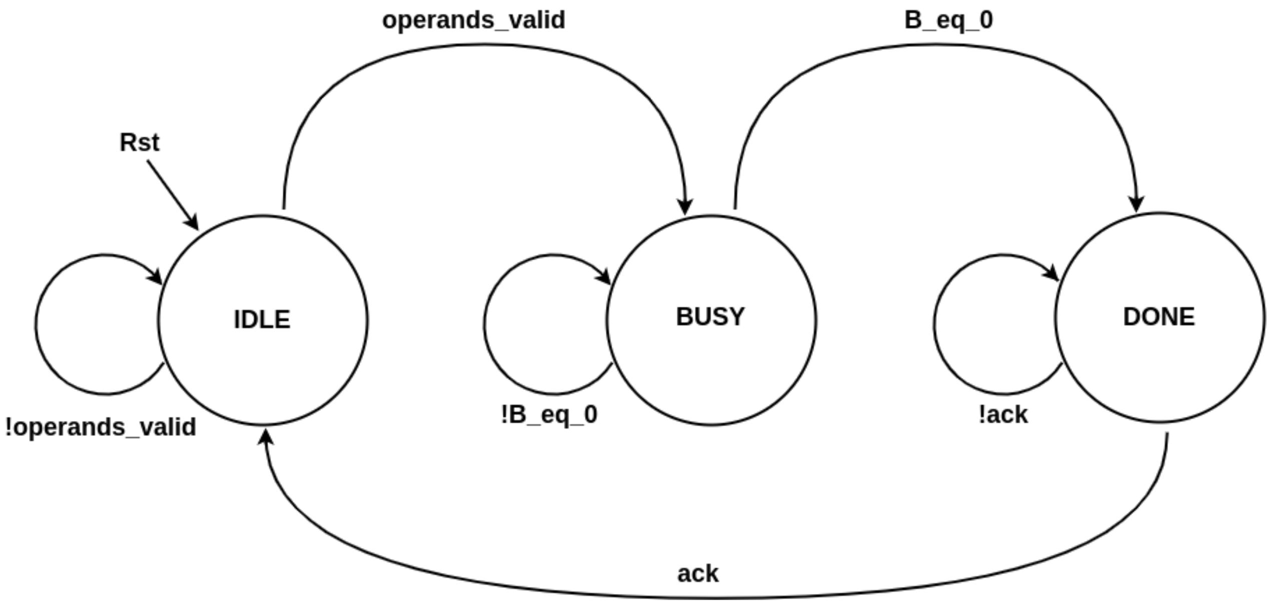

Figure 2: FSM for the states of the system

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.

Testbench Architecture

The testbench consists of the following main components:

- **Interface** - `gcd_if`: Provides a structured connection between testbench and DUT.
- **Transaction Class** - `gcd_packet`: Stores stimulus operands and manages randomization.
- **Golden Model Function** - `calc_gcd()`: Computes reference GCD values.
- **Driver** - Sends transactions to the DUT through the interface.
- **Monitor/Scoreboard** - Verifies DUT outputs by comparing them with expected results.
- **Testbench Top Module** - Instantiates the DUT and coordinates all verification components.

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.

```
// GCD Interface
interface gcd_if(input bit Clk);
    logic [7:0] A_in;
    logic [7:0] B_in;
    logic operands_val;
    logic ack;
    logic [7:0] gcd_out;
    logic gcd_valid;
    logic Rst;

    // Clocking blocks
    clocking cb @(posedge Clk);
        output A_in, B_in, operands_val, ack, Rst;
        input gcd_out, gcd_valid;
    endclocking
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 in SystemVerilog is responsible for stimulating the Device Under Test (DUT) by interacting with the **gcd_if** virtual interface. It is constructed using a virtual interface handle **vif** and an input queue **input_queue**, which contains **gcd_packet** objects representing test vectors.

The key functionality lies in the **run()** task. It begins by asserting the **Rst** signal to reset the DUT and waits for two clock cycles before deasserting it. The driver then enters a loop where it processes packets from the input queue one by one. For each packet, it drives the operands **A_in** and **B_in** onto the interface and asserts the **operands_val** signal to indicate valid input. After one clock cycle, **operands_val** is deasserted. The driver waits for the DUT to assert **gcd_valid**, signaling that the GCD computation is complete. It then asserts **ack** to acknowledge receipt of the result before moving to the next packet.

The **display_queue()** task is used for debugging, printing the contents of the input queue and allowing verification of which packets are pending transmission. Once the queue is empty, the driver task waits for a few additional cycles before terminating. This class provides a clean and systematic way of applying stimulus to the DUT and managing handshake signals in the verification environment.

```systemverilog
// Driver Class
class driver;
    virtual gcd_if vif;
    gcd_packet input_queue[$];

    function new(virtual gcd_if vif, gcd_packet in_q[$]);
        this.vif = vif;
        this.input_queue = in_q;  // Just copy the queue
    endfunction


    task display_queue(string header = "[DRIVER] Input Queue:");
      $display("%s (Size: %0d)", header, input_queue.size());

      foreach (input_queue[i]) begin
          $display("  Packet[%0d] --> A = %0d, B = %0d, operands_valid = %0d",
                   i, input_queue[i].A, input_queue[i].B, input_queue[i].operands_valid);
      end
        endtask

    task run();
        gcd_packet pkt;

        vif.cb.Rst <= 1;
        repeat (2) @(vif.cb);  // Reset pulse
        vif.cb.Rst <= 0;

        forever begin
            if (!input_queue.empty()) begin
                pkt = input_queue.pop_front();
                // display_queue("Queue After Pop:");  //Used for debuging

                // Wait random time before sending operands
                repeat ($urandom_range(1, 5)) @(vif.cb);

                vif.cb.A_in <= pkt.A;
                vif.cb.B_in <= pkt.B;
                vif.cb.operands_val <= pkt.operands_valid;
                // $display("[%0t] DRIVER: Sent operands A = %0d, B = %0d, asserted operands_val", $time, pkt.A, pkt.B);//Used for debugging

                @(vif.cb);  // One clock

                vif.cb.operands_val <= 0;  // De-assert after sending
                @(vif.cb);

                wait (vif.gcd_valid == 1);

                repeat ($urandom_range(1, 5)) @(vif.cb);
```

```
                    vif.cb.ack <= 1;
                   @(vif.cb);
                    vif.cb.ack <= 0;

               end else begin
                    // Input queue is empty, exit task after a few cycles (
                       optional wait)
                    $display("[%0t] DRIVER: Input queue empty, finishing..."
                       , $time);
                    repeat (5) @(vif.cb);
                    break;
               end
          end

          $display("[%0t] DRIVER: Task completed!", $time);
     endtask

endclass
```

Monitor and Scoreboard

The **monitor** class in SystemVerilog is responsible for verifying the outputs of the Device Under Test (DUT). It connects to the DUT through the **gcd_if** virtual interface and maintains an **expect_queue**, which stores the expected Greatest Common Divisor (GCD) results for each transaction.

The main functionality is implemented in the **run()** task. The monitor continuously observes the **gcd_valid** signal from the DUT. When **gcd_valid** transitions from 0 to 1, it means the DUT has produced a new GCD output. At this point, the monitor retrieves the next expected value from the **expect_queue** and compares it with the DUT's output signal **gcd_out**. If they match, it prints a [**PASS**] message; otherwise, it reports a [**FAIL**] message, showing both the DUT's output and the expected value for debugging.

The monitor continues this process until all expected results have been checked and the **expect_queue** becomes empty. After a few additional clock cycles, the task concludes. This class effectively serves as a basic scoreboard, providing automated result checking within the testbench environment.

```
// Monitor and Scoreboard
class monitor;
     virtual gcd_if vif;
     int expect_queue[$];

     function new(virtual gcd_if vif, ref int exp_q[$]);
          this.vif = vif;
          this.expect_queue = exp_q;
     endfunction

     task run();
     int expected;
     bit prev_gcd_valid = 0;

```

```
    forever begin
        @(posedge vif.Clk);

        if (vif.gcd_valid == 1 && prev_gcd_valid == 0) begin
            if (expect_queue.empty()) begin
                $display("[%0t] MONITOR: Expect queue empty, nothing to check!", $time);
            end else begin
                expected = expect_queue.pop_front();

                if (vif.gcd_out == expected) begin
                    $display("[PASS] Time=%0t | DUT: %0d Expected: %0d",
                              $time, vif.gcd_out, expected);
                end else begin
                    $display("[FAIL] Time=%0t | DUT: %0d Expected: %0d",
                              $time, vif.gcd_out, expected);
                end
            end
        end

        prev_gcd_valid = vif.gcd_valid;

        // Exit when queue is empty
        if (expect_queue.empty()) begin
            $display("[%0t] MONITOR: All transactions checked, finishing...", $time);
            repeat (5) @(posedge vif.Clk);
            break;
        end
    end

    $display("[%0t] MONITOR: Task completed!", $time);
        endtask
endclass
```

Testbench Top Module

The **tb_top** module is the top-level testbench for verifying the GCD hardware design. It orchestrates the simulation by generating the clock, initializing signals, and connecting all verification components with the Device Under Test (DUT).

A clock signal **Clk** toggles every 5 time units, providing a 100MHz clock frequency. The **gcd_if** interface is instantiated and linked to this clock, facilitating communication between the testbench components and the DUT.

Two queues are defined:

- **input_q**: A queue of **gcd_packet** transactions, representing randomized inputs to the DUT.
- **expect_q**: A queue of integers holding the expected GCD results calculated using the golden model function **calc_gcd()**.

The DUT, represented by **top_module**, is instantiated and connected through the interface signals and clock.

Within the **initial** block:

1. The clock signal is initialized to zero.
2. Informational messages mark the start of the simulation.
3. Ten randomized **gcd_packet** transactions are generated. Each packet is added to **input_q**, and its corresponding expected GCD is computed and pushed into **expect_q**.
4. Instances of the **driver** and **monitor** classes are created, receiving the interface and queues as arguments.
5. The **driver.run()** and **monitor.run()** tasks are executed in parallel using a **fork...join** block.
6. Upon completion of all tasks, messages indicate the end of the simulation, and the simulation is terminated with **$finish**.

This module ensures that randomized test scenarios are applied to the DUT and verifies the correctness of its outputs through automated checking.

```
// Testbench Top
module tb_top;

    // Clock and reset
    bit Clk;
    always #5 Clk = ~Clk;  // 100MHz clock

    // Interfaces
    gcd_if gcd_vif(Clk);

    // Queues
    gcd_packet input_q[$];
    int expect_q[$];

    // DUT Instance
    top_module dut (
        .A_in(gcd_vif.A_in),
        .B_in(gcd_vif.B_in),
        .operands_val(gcd_vif.operands_val),
        .Clk(Clk),
        .Rst(gcd_vif.Rst),
        .ack(gcd_vif.ack),
        .gcd_out(gcd_vif.gcd_out),
        .gcd_valid(gcd_vif.gcd_valid)
    );


    // Driver & Monitor Instances
    driver drv;
    monitor mon;

    initial begin

    Clk = 0;

    $display("**********SIMULATION STARTED*********");

    // Generate transactions
    repeat (10) begin
        gcd_packet pkt = new();
        pkt.randomize();
        pkt.display("[TB] Generated packet ->");
      input_q.push_back(pkt.copy());

        expect_q.push_back(calc_gcd(pkt.A, pkt.B));  // Directly push
           the result

    end

    $display("********PACKETS GENERATED*************");

    // Create driver and monitor
    drv = new(gcd_vif, input_q);
    mon = new(gcd_vif, expect_q);

    // Fork parallel execution of driver and monitor
    fork
      drv.run();
      mon.run();
    join // Run the simulation till all the active threads are finished
       (yaaay)

    $display("*********SIMULATION COMPLETED************");

    $finish;

  end
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