Interfaces, Clocking Blocks & Race-Free Verification

Introduction

With the completion of the earlier phases:

- **Day 1** explained *why* modern verification needs structure
- **Day 2** explained *how* SystemVerilog models data and execution
- **Day 3** explained *how behavior is abstracted and reused over time*
-

Day 4 marks a critical turning point.
This phase focuses on **how the testbench safely connects to the DUT**, controls timing, and avoids subtle race conditions that silently break verification. Many verification bugs are not design bugs they are **testbench–DUT synchronization bugs**.

Scope of Day 4

Day 4 focuses on **testbench–DUT interaction and timing control**, covering:

- Interfaces
- Modports
- Clocking blocks
- Program blocks
- Race conditions
- Basic assertions

These concepts are **mandatory** before moving into:

- Constrained-random stimulus
- Coverage-driven verification
- UVM

1. Interface:

An **interface** is a container that groups related signals and optionally behavior. Instead of connecting many individual wires, the testbench and DUT connect using **a single interface instance**.

1.1 Why Interfaces Exist

Interfaces solve multiple problems:

- Eliminate long port lists
- Group related signals logically
- Centralize timing and direction
- Improve readability and reuse

Fig. 1 : Connection of DUT and TESTBENCH via INTERFACE

The interface is **not just a bundle of wires** — it is a **formal communication boundary**.

What Actually Happens Without an Interface

In traditional Verilog-style testbenches:

- Every signal is connected individually
- Large port lists are required
- Signal direction is implicit
- Timing is scattered across modules

As designs grow, this results in:

- Poor readability
- High maintenance effort
- Increased risk of race conditions
- Difficult debugging

What the Interface Changes

With a SystemVerilog interface:

- All related signals are **grouped in one place**
- The DUT and testbench both connect to the **same interface instance**
- Timing and direction can be defined **centrally**
- The interface becomes the **single source of truth** for communication

Conceptually:
The DUT does not talk directly to the testbench. Both talk to the interface.

Simple Interface Example Program

We will take the example of a 2-bit adder to understand how a Design Under Test (DUT) is connected to a testbench using an interface.

```
// 2 bit Adder code (DUT design)  adder.v

module adder (
  input  [1:0] a,
  input  [1:0] b,
  input        cin,
  output [1:0] sum,
  output       cout
);

  assign {cout, sum} = a + b + cin;

endmodule
```

The above module represents a simple 2-bit adder. It takes two 2-bit inputs a and b, a carry-in cin, and produces a 2-bit sum and a carry-out cout.

Interface Design

To design an interface, we use the **interface** keyword. All the signals associated with the DUT are declared inside the interface so that they can be shared between the DUT and the testbench.

```
 // adder_if.sv
interface adder_if;
  logic [1:0] a, b;
  logic       cin;
  logic [1:0] sum;
  logic       cout;
endinterface
```

The interface groups all input and output signals of the DUT, reducing direct signal handling in the testbench.

// Test Bench

```
module adder_tb;

  //  We instantiate interface module adder_if, so that we can use those variables instead of directly
using DUT variables.

  adder_if intf();
```

```
  // DUT instance. We call interface variables to instatiate DUT module, instead of direct variables

  adder DUT (
    .a   (intf.a),
    .b   (intf.b),
    .cin (intf.cin),
    .sum (intf.sum),
    .cout(intf.cout)
  );

  initial begin
    intf.a   = 2'b01;
    intf.b   = 2'b10;
    intf.cin = 1'b0;
    #10;

    intf.a   = 2'b11;
    intf.b   = 2'b01;
    intf.cin = 1'b1;
    #10;

    $finish;
  end

  initial begin
    $monitor("T=%0t a=%b b=%b cin=%b => sum=%b cout=%b",
             $time, intf.a, intf.b, intf.cin, intf.sum, intf.cout);
  end

endmodule
```

Output :

```
# T=0 a=01 b=10 cin=0 => sum=11 cout=0
# T=10 a=11 b=01 cin=1 => sum=01 cout=1
```

In the testbench, the interface is instantiated and its signals are connected to the DUT ports. The testbench drives inputs through the interface and observes outputs through the same interface.

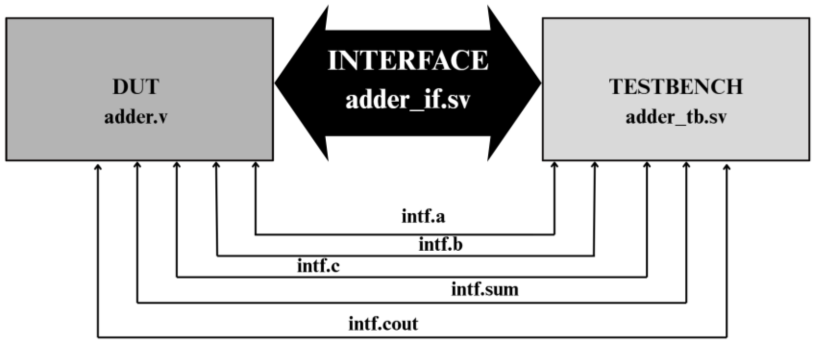

Fig 2.2 Bit Adder design using Interface

2. Modports

2.1 Why Modports Are Needed

Without direction control:

- Both DUT and testbench can drive the same signal
- Accidental multiple drivers occur
- Bugs appear silently

Modports define signal direction per user.

To understand modport, we will take same dut 2 bit adder.

```
// 2 bit Adder code (DUT design)  adder.v

module adder (
  input  [1:0] a,
  input  [1:0] b,
  input        cin,
  output [1:0] sum,
  output       cout
);

  assign {cout, sum} = a + b + cin;

endmodule
```

The above module represents a simple 2-bit adder. It takes two 2-bit inputs a and b, a carry-in cin, and produces a 2-bit sum and a carry-out cout.

Interface with Modports

The interface groups all DUT signals. Using **modports**, we define **directional access** for the DUT and the testbench. Keyword to use modports is **modport.**

```
// adder_if.sv
interface adder_if;

  logic [1:0] a, b;
  logic       cin;
  logic [1:0] sum;
  logic       cout;

  // Modport for DUT
  modport DUT_MP (
    input  a, b, cin,
    output sum, cout
  );

  // Modport for Testbench
  modport TB_MP (
    output a, b, cin,
    input  sum, cout
  );

endinterface
```

Testbench Using Modports

```
module adder_tb;

  // Interface instance
  adder_if intf();

  // DUT instantiation using modport
  adder DUT (
    .a   (intf.DUT_MP.a),
    .b   (intf.DUT_MP.b),
    .cin (intf.DUT_MP.cin),
    .sum (intf.DUT_MP.sum),
    .cout(intf.DUT_MP.cout)
  );

  // Apply stimulus via TB modport
  initial begin
```

```
    intf.TB_MP.a   = 2'b01;
    intf.TB_MP.b   = 2'b10;
    intf.TB_MP.cin = 1'b0;
    #10;

    intf.TB_MP.a   = 2'b11;
    intf.TB_MP.b   = 2'b01;
    intf.TB_MP.cin = 1'b1;
    #10;

    $finish;
  end

  // Monitor outputs
  initial begin
    $monitor("T=%0t a=%b b=%b cin=%b => sum=%b cout=%b",
             $time,
             intf.TB_MP.a,
             intf.TB_MP.b,
             intf.TB_MP.cin,
             intf.TB_MP.sum,
             intf.TB_MP.cout);
  end

endmodule
```

Output :

```
# T=0 a=01 b=10 cin=0 => sum=11 cout=0
# T=10 a=11 b=01 cin=1 => sum=01 cout=1
```

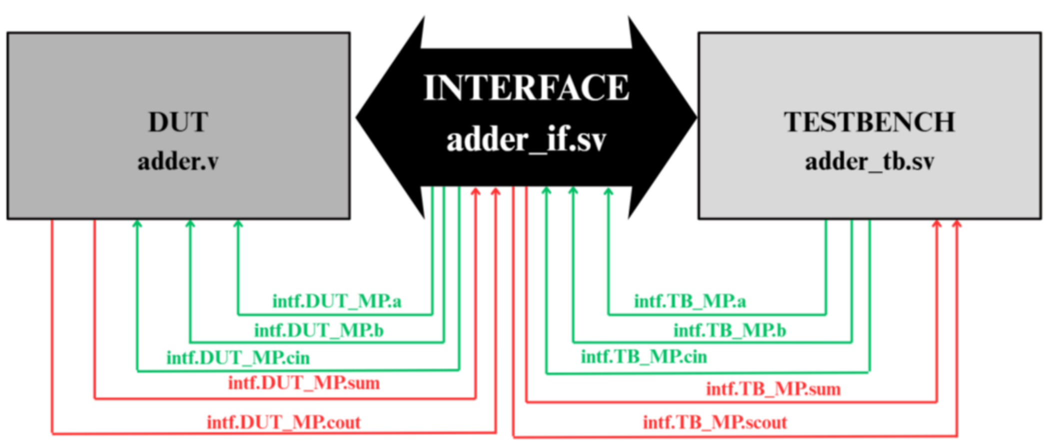

Fig 3. DUT Design using Modports in interface

3. Clocking Block

3.1 The Problem Without Clocking Blocks

Without using clocking blocks:
- The **testbench and DUT execute in the same simulation time slot**
- Both may **read and drive signals at the same time**
- This can lead to **race conditions**
- Simulation results may become **tool or simulator dependent**

This issue is common in synchronous designs when both DUT and testbench are triggered on the same clock edge.

3.2 What Is a Clocking Block?

A **clocking block** is a SystemVerilog construct used inside an interface that:
- Defines **when signals are driven and sampled**
- Separates **testbench timing** from **DUT timing**
- Provides **skew control** for inputs and outputs
- Eliminates **race conditions** between DUT and testbench

We use same DUT for understanding clocking blocks

```
// 2 bit Adder code (DUT design)  adder.v

module adder (
  input  [1:0] a,
  input  [1:0] b,
  input        cin,
  output [1:0] sum,
  output       cout
);

  assign {cout, sum} = a + b + cin;

endmodule
```

The above module represents a simple 2-bit adder. It takes two 2-bit inputs a and b, a carry-in cin, and produces a 2-bit sum and a carry-out cout.

Interface Design using Clocking blocks

```
// adder_if.sv
interface adder_if (input logic clk);

  logic [1:0] a, b;
  logic       cin;
  logic [1:0] sum;
  logic       cout;

  // Clocking block for testbench
  clocking cb @(posedge clk);
    output a, b, cin;   // Driven by testbench
    input  sum, cout;  // Sampled from DUT
  endclocking

endinterface
```

The clocking block ensures that:

- Inputs (a, b, cin) are driven **after** the clock edge
- Outputs (sum, cout) are sampled **after** the DUT updates

Using Clocking Block in Testbench

```
module adder_tb;

  logic clk;
  always #5 clk = ~clk;

  // Interface instance
  adder_if intf(clk);

  // DUT instance
  adder DUT (
    .a   (intf.a),
    .b   (intf.b),
    .cin (intf.cin),
    .sum (intf.sum),
    .cout(intf.cout)
  );

  initial begin
    clk = 0;

    // Apply stimulus using clocking block
```

```
    intf.cb.a   <= 2'b01;
    intf.cb.b   <= 2'b10;
    intf.cb.cin <= 1'b0;
    ##1;

    $display("sum=%b cout=%b", intf.cb.sum, intf.cb.cout);

    intf.cb.a   <= 2'b11;
    intf.cb.b   <= 2'b01;
    intf.cb.cin <= 1'b1;
    ##1;

    $display("sum=%b cout=%b", intf.cb.sum, intf.cb.cout);

    $finish;
  end

endmodule
```

Key Advantage of Clocking Blocks

- Race-free signal access
- Clear separation of DUT and testbench timing
- Cycle-accurate verification
- Predictable simulation behavior

4. **Race Conditions**

4.1 What Is a Race Condition?

A race occurs when:

- DUT and testbench access the same signal
- At the same simulation time
- Order is undefined

4.2 Why Races Are Dangerous

- Test passes on one simulator
- Fails on another
- Bugs disappear or appear randomly

How SystemVerilog Prevents Races

Feature	Purpose
Interface	Signal grouping
Modport	Direction control
Clocking block	Timing separation

Key Takeaways – Day 4

- Interfaces organize DUT–testbench connectivity
- Modports enforce direction safety
- Clocking blocks eliminate race conditions
- Race-free testbenches are **mandatory**, not optional

A correct DUT with a broken testbench is still a failed verification.

What Comes Next?

Object-Oriented Programming (OOPS) Concepts