More insight on Testbench

From Simple Verilog to Structured Verification

Features

- System task
- File interaction
- Applying stimulus at signal transition

System task

- A **system task** is a **built-in command provided by the simulator**, not

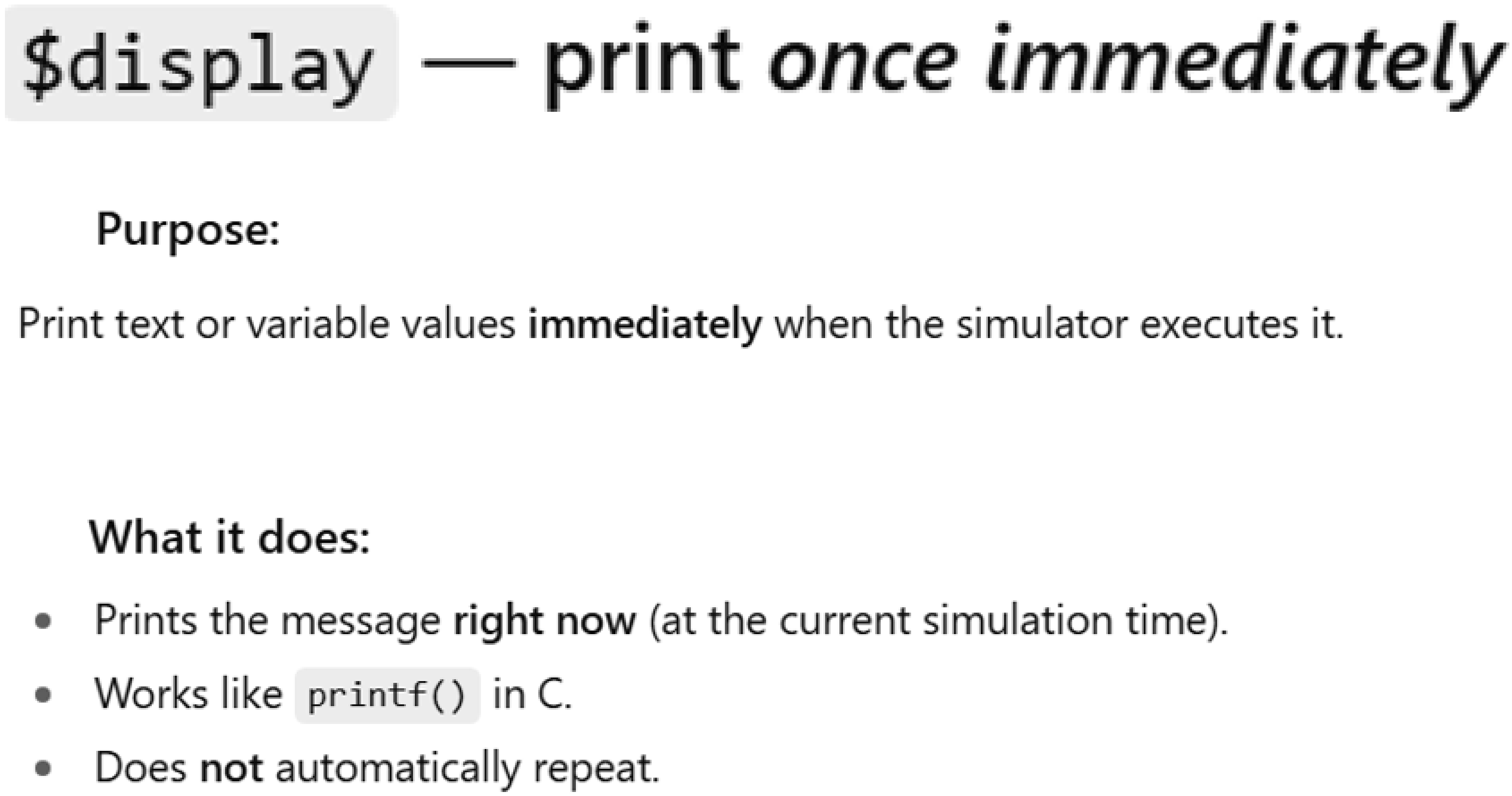
  synthesized into hardware.

- They **control or observe the simulation**, but **do not describe real hardware**.

- All system tasks start with a **$ symbol** (dollar sign) —

- They are part of the **simulation system**, not the circuit.

Most useful system tasks in Verilog TB



`$display` — print *once immediately*

Purpose:

Print text or variable values **immediately** when the simulator executes it.

What it does:

- Prints the message **right now** (at the current simulation time).
- Works like `printf()` in C.
- Does **not** automatically repeat.

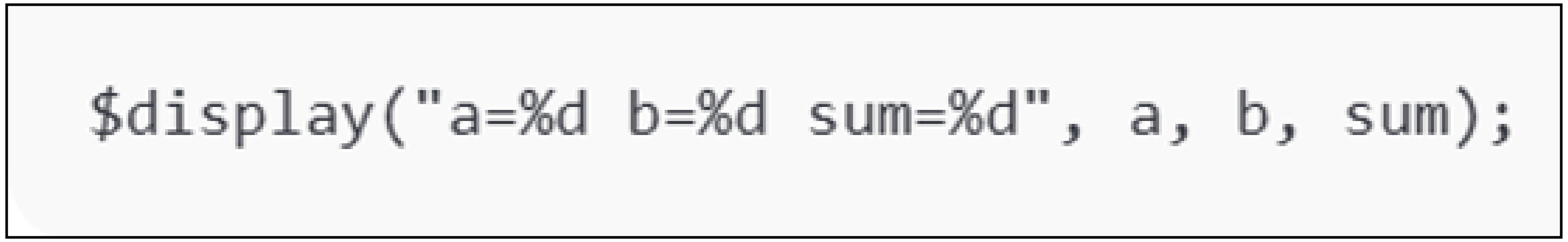
`$display` format basics

`$display` lets you print variables in a formatted way:

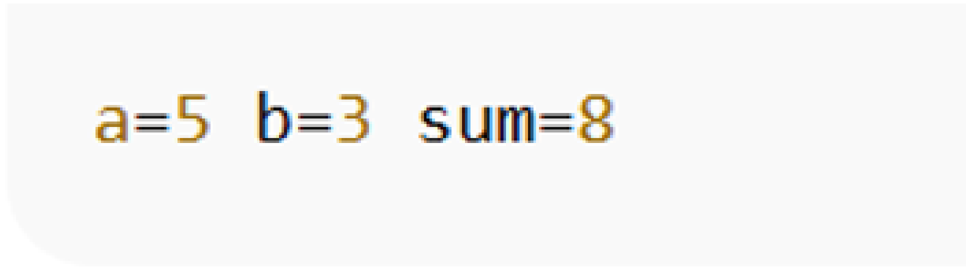
```verilog
$display("a=%d b=%d sum=%d", a, b, sum);
```

Here:

- `%d` → print as **decimal**
- `%b` → print as **binary**
- `%h` → print as **hexadecimal**
- `%t` → print **simulation time**
- `%0t` or `%4t` → print time with **field width formatting**



```
$display("a=%d b=%d sum=%d", a, b, sum);
```



```
a=5 b=3 sum=8
```

`$monitor` — print *automatically on any signal change*

Keep watching one or more signals, and print **every time** any of them changes.

What it does:

- Prints a line **whenever** any listed signal changes value.
- Runs **continuously** until simulation ends (or you turn it off).

```
$monitor("t=%0t clk=%b rst=%b a=%d b=%d sum=%d",
         $time, clk, rst, a, b, sum);
```

```
t=0 clk=0 rst=1 a=0 b=0 sum=0
t=15 clk=1 rst=0 a=0 b=0 sum=0
t=25 clk=1 rst=0 a=5 b=3 sum=8
t=35 clk=1 rst=0 a=10 b=7 sum=17
```

What it does:

- Prints a line **whenever** any listed signal changes value.
- Runs **continuously** until simulation ends (or you turn it off).
- You only write it **once** — no loops or delays needed.

`$finish` — end the simulation

Purpose:

Stops simulation completely and closes all output files or waveforms.

Example:

```verilog
$finish;
```

What it does:

- Tells the simulator “I’m done — stop now.”
- Used at the end of your testbench (after all vectors applied).

```verilog
initial begin
  $display("Simulation started at t=%0t", $time);
  $monitor("t=%0t a=%0d b=%0d sum=%0d", $time, a, b, sum);

  a=5; b=3;
  #10 a=10; b=7;
  #10 a=255; b=1;

  #10;
  $display("Simulation finished at t=%0t", $time);
  $finish;
end
```

🗒 Console Output:

```
Simulation started at t=0
t=0 a=5 b=3 sum=8
t=10 a=10 b=7 sum=17
t=20 a=255 b=1 sum=256
Simulation finished at t=30
```

```
initial begin
  a = 0; b = 0;
  #10 a = 5; b = 3;  $display("Time=%0t a=%0d b=%0d sum=%0d", $time,a,b,sum);
  #10 a = 10; b = 7; $display("Time=%0t a=%0d b=%0d sum=%0d", $time,a,b,sum);
  #10 a = 255; b = 1;$display("Time=%0t a=%0d b=%0d sum=%0d", $time,a,b,sum);
end
```

```
initial begin
  $monitor("Time=%0t a=%0d b=%0d sum=%0d", $time, a, b, sum);
  a = 0; b = 0;
  #10 a = 5; b = 3;
  #10 a = 10; b = 7;
  #10 a = 255; b = 1;
  #10 $finish;
end
```

Typical separation in **testbenches**:

Block Type	Typical Purpose
`always` block	Generate clock: `always #5 clk = ~clk;`
`initial` block #1	Apply reset
`initial` block #2	Apply test stimulus
`initial` block #3	Monitor or finish simulation

Test Stimulus

- Case-I: # delay & explicit signal value assignment for specific test cases

  (Can we provide the stimulus for all possible test cases).

- Case-II: # delay & signal value come from file (result may be stored in file)

- Case-III: No # delay, but signal value (explicitly) would be changed before every positive edge

- Case-IV: No # delay & signal value come from file & signal value (explicitly) would be changed before every positive edge

Question

Write Verilog code for an 8-bit parallel adder.

The adder should take two 8-bit numbers as input (`a`, `b`) and produce a 9-bit sum output.

```
// File: adder.v
module adder (
    input  [7:0] a,      // 8-bit input A
    input  [7:0] b,      // 8-bit input B
    output [8:0] sum     // 9-bit output (to hold carry)
);
    assign sum = a + b;  // simple combinational addition
endmodule
```

Write a testbench for the 8-bit adder

“Apply the following input pairs to a and b at 0 ns, 10 ns, 20 ns, and 30 ns respectively:(5, 3), (10, 7), (100, 55), (255, 1).”

```
// File: tb_adder.v
`timescale 1ns/1ps
module tb_adder;

  // Declare signals to connect to DUT
  reg   [7:0] a, b;
  wire  [8:0] sum;

  // Instantiate the DUT
  adder uut (
    .a(a),
    .b(b),
    .sum(sum)
  );

  // Stimulus block
  initial begin
    // Apply a few test cases
    a = 8'd5;    b = 8'd3;     #10;

    a = 8'd10;   b = 8'd7;     #10;

    a = 8'd100;  b = 8'd55;    #10;

    a = 8'd255;  b = 8'd1;     #10;

    // Finish simulation
    $finish;
  end

endmodule
```

Explicit time (delay) and the value of the signals

Q. Write Verilog code for a *registered 8-bit parallel adder* and its testbench.

The circuit should add two 8-bit inputs `a` and `b`, and store the result in a 9-bit register `sum` on every **positive edge of the clock**.

```verilog
// File: adder_reg.v
module adder_reg (
    input          clk,
    input          rst,          // active high reset
    input   [7:0] a,
    input   [7:0] b,
    output reg [8:0] sum         // registered output
);
    always @(posedge clk or posedge rst) begin
        if (rst)
            sum <= 9'd0;         // reset sum to 0
        else
            sum <= a + b;        // store sum on clock edge
    end
endmodule
```

Write a testbench for the 8-bit adder

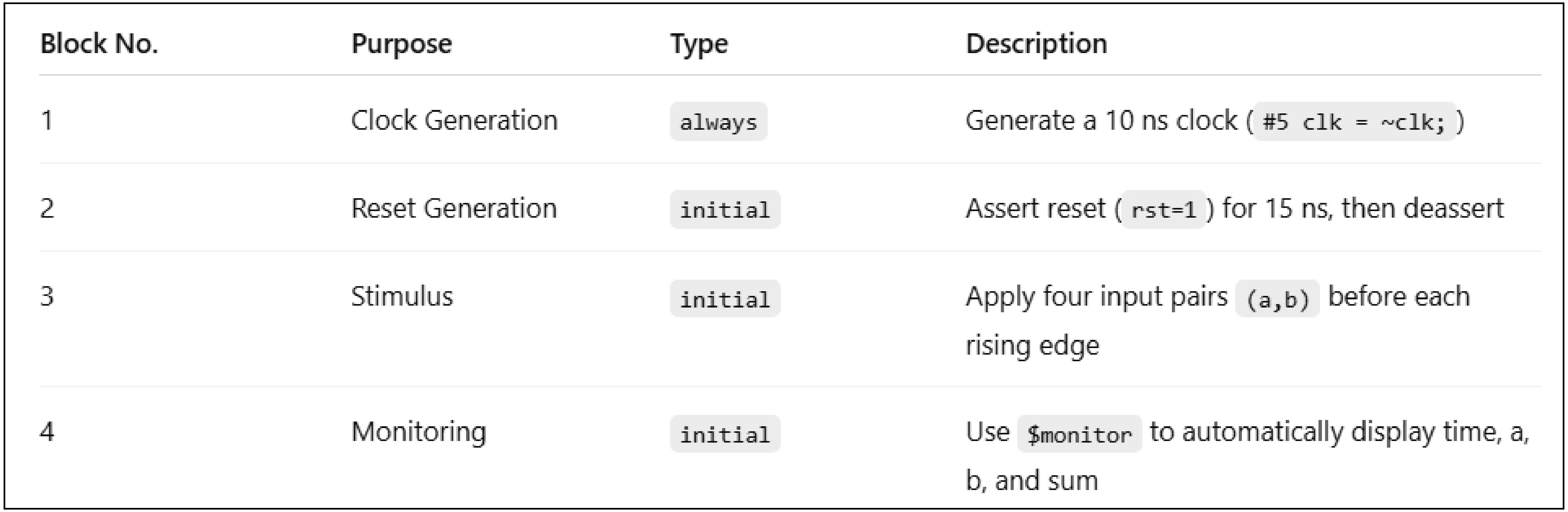
- **Clk period = 10 ns**
- **rst should be high initially till t=15 ns**
- **Apply the following input pairs to a and b at 0 ns, 20 ns, 40 ns, and 60 ns respectively:(5, 3), (10, 7), (255, 1).”**

```
`timescale 1ns/1ps
module tb_adder_reg;
  reg clk, rst;
  reg [7:0] a, b;
  wire [8:0] sum;
  // Instantiate DUT
  adder_reg UUT (.clk(clk), .rst(rst), .a(a), .b(b), .sum(sum));

  // Clock generation block

  initial clk = 0;
  always #5 clk = ~clk;   // 10 ns period

  // Reset block (separate)
  initial begin
    rst = 1;                        // assert reset
    #15 rst = 0;                    // deassert reset after 15 ns
  end
  // Stimulus block
  initial begin
    a = 8'd00; b = 8'd0;  #20;
    a = 8'd5;   b = 8'd3; #20;
    a = 8'd10;  b = 8'd7; #20;
    a = 8'd255; b = 8'd1; #20;
    #20 $finish;                    // end simulation
  end

  // APPLY MONITOR (one line; auto-prints on any change)
  initial
    $monitor("t=%0t  clk=%b  rst=%b  a=%0d  b=%0d  sum=%0d",
             $time,  clk,     rst,     a,      b,      sum);
endmodule
```

```
// APPLY MONITOR (one line; auto-prints on any change)
initial
  $monitor("t=%0t  clk=%b  rst=%b  a=%0d  b=%0d  sum=%0d",
           $time,  clk,      rst,     a,      b,      sum);
```

Four separate blocks:

Each serving one specific purpose



Block No.	Purpose	Type	Description
1	Clock Generation	`always`	Generate a 10 ns clock (`#5 clk = ~clk;`)
2	Reset Generation	`initial`	Assert reset (`rst=1`) for 15 ns, then deassert
3	Stimulus	`initial`	Apply four input pairs `(a,b)` before each rising edge
4	Monitoring	`initial`	Use `$monitor` to automatically display time, a, b, and sum

```
//=========================================================
// 1. Clock generation block
//=========================================================
initial clk = 0;
always #5 clk = ~clk;     // 10 ns clock period

//=========================================================
// 2. Reset generation block
//=========================================================
initial begin
  rst = 1'b1;             // Assert reset at start
  #15 rst = 1'b0;         // Deassert reset after 15 ns
end
```

```
//==========================================================
// 3. Stimulus block
//==========================================================
```

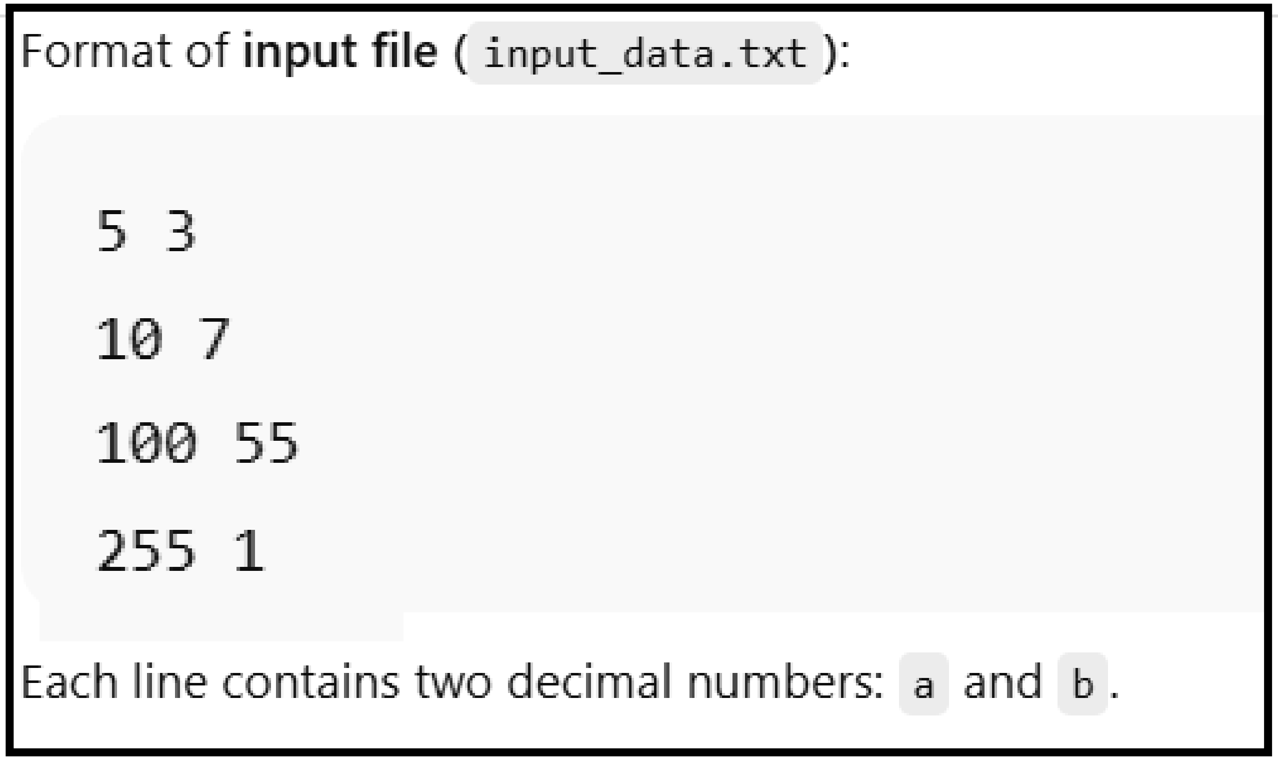
```
   initial begin
     rst = 1;                      // assert reset
     #15 rst = 0;                  // deassert reset after 15 ns
   end
   // Stimulus block
   initial begin
     a = 8'd00; b = 8'd0;  #20;
     a = 8'd5;   b = 8'd3; #20;
     a = 8'd10;  b = 8'd7; #20;
     a = 8'd255; b = 8'd1; #20;
     #20 $finish;                  // end simulation
   end
```

```
//=========================================================
// 4. Monitor block
//=========================================================
initial begin
  $monitor("t=%0t clk=%b rst=%b a=%0d b=%0d sum=%0d",
           $time, clk, rst, a, b, sum);
end
```

File-Based Testbench for Registered 8-bit Adder

Q. Write a Verilog testbench for the registered 8-bit adder

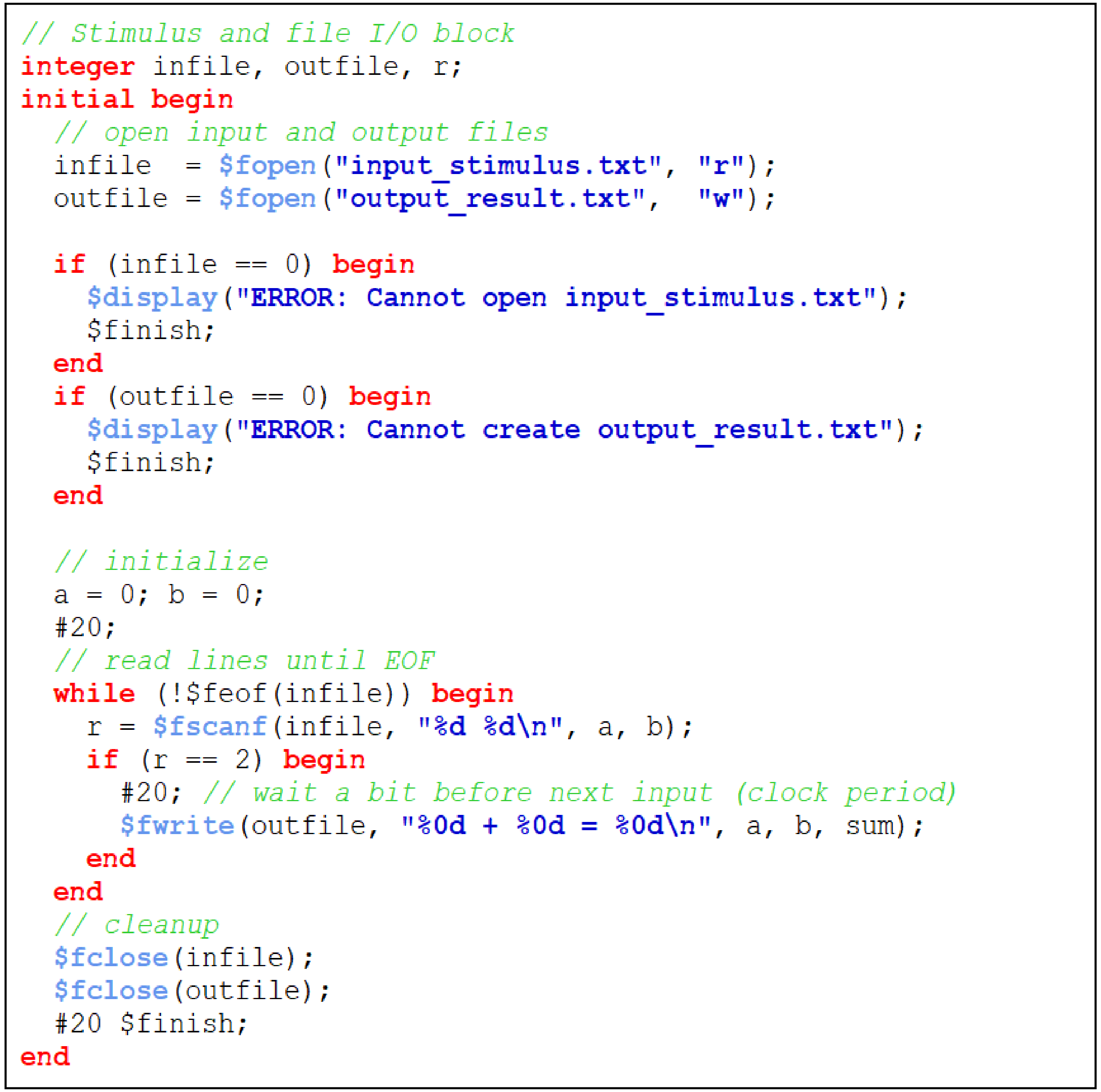
that takes its input values from a text file and writes the results to another output file.



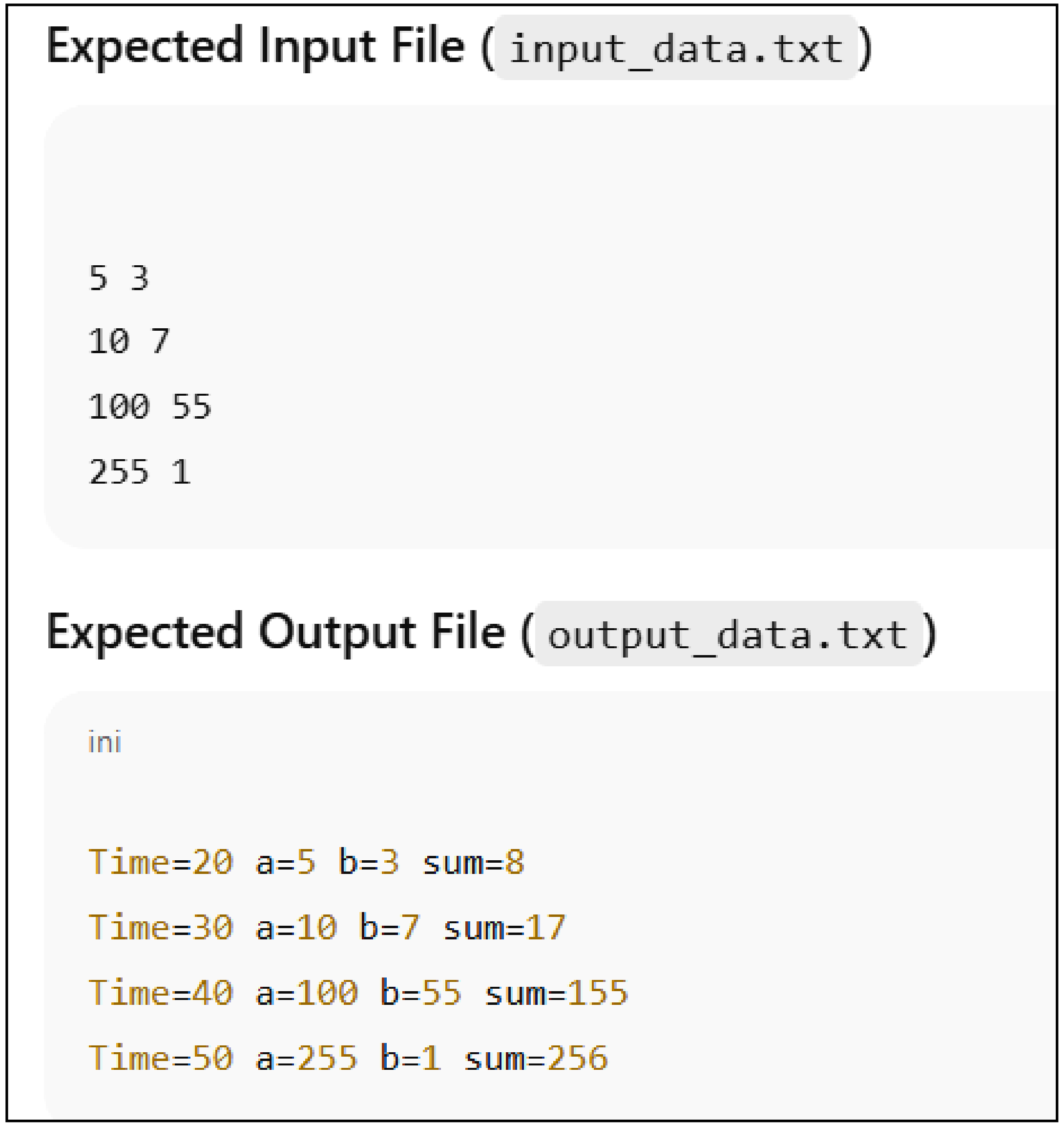
Format of **input file** (`input_data.txt`):

```
5 3
10 7
100 55
255 1
```

Each line contains two decimal numbers: `a` and `b`.



```
// Stimulus and file I/O block
integer infile, outfile, r;
initial begin
  // open input and output files
  infile  = $fopen("input_stimulus.txt", "r");
  outfile = $fopen("output_result.txt",  "w");

  if (infile == 0) begin
    $display("ERROR: Cannot open input_stimulus.txt");
    $finish;
  end
  if (outfile == 0) begin
    $display("ERROR: Cannot create output_result.txt");
    $finish;
  end

  // initialize
  a = 0; b = 0;
  #20;
  // read lines until EOF
  while (!$feof(infile)) begin
    r = $fscanf(infile, "%d %d\n", a, b);
    if (r == 2) begin
      #20; // wait a bit before next input (clock period)
      $fwrite(outfile, "%0d + %0d = %0d\n", a, b, sum);
    end
  end
  // cleanup
  $fclose(infile);
  $fclose(outfile);
  #20 $finish;
end
```



Expected Input File (`input_data.txt`)

```
5 3

10 7

100 55

255 1
```

Expected Output File (`output_data.txt`)

```ini
Time=20 a=5 b=3 sum=8

Time=30 a=10 b=7 sum=17

Time=40 a=100 b=55 sum=155

Time=50 a=255 b=1 sum=256
```

TestBench

Write a Verilog testbench for the registered 8-bit adder such that the inputs a and b are changed only at the negedge of the clock, ensuring the DUT samples them on the next posedge.

Simulation event	Clock edge	Inputs applied	Expected action
After reset (≈15 ns)	`negedge clk`	a=5, b=3	Will be sampled at next posedge
Next negedge	a=10, b=7	—	
Next negedge	a=100, b=55	—	
Next negedge	a=255, b=1	—	

```verilog
//========================================
// 3. Stimulus (inputs change at negedge clk)
//========================================
initial begin
  a = 0; b = 0;
  @(negedge rst);           // Wait for reset release

  @(negedge clk); a = 8'd5;   b = 8'd3;
  @(negedge clk); a = 8'd10;  b = 8'd7;
  @(negedge clk); a = 8'd100; b = 8'd55;
  @(negedge clk); a = 8'd255; b = 8'd1;

  #20;
  $finish;
end
```

```
// stimulus: change inputs at negedge so they're stable before posedge
initial begin
  a = 0; b = 0;
  @(negedge rst);

  @(negedge clk); a = 8'd5;   b = 8'd3;
  @(negedge clk); a = 8'd10;  b = 8'd7;
  @(negedge clk); a = 8'd255; b = 8'd1;

  #20;
  $finish;
end
```

We can automatically

provide all possible input

stimuli for a and b

```
// Stimulus block (independent)
integer i, j;
initial begin
  a = 0; b = 0;
  @(negedge rst);

  // Simple nested loop of stimuli

   for (i = 0; i < 256; i = i + 1) begin
     for (j = 0; j < 256; j = j + 1) begin
       @(negedge clk);
       a = i[7:0];
       b = j[7:0];
     end
   end


  #20;
  $finish;
end
```

```
// File I/O and stimulus
integer infile, outfile, r;

initial begin
  infile  = $fopen("input_stimulus.txt", "r");
  outfile = $fopen("output_result.txt",  "w");

  if (infile == 0 || outfile == 0) begin
    $display("ERROR: File open failed");
    $finish;
  end

  // Initialize
  a = 0; b = 0;
  #20;                         // wait till reset completes

  // Loop through input file
  while (!$feof(infile)) begin
    r = $fscanf(infile, "%d %d\n", a, b);
    if (r == 2) begin
      @(negedge clk);          // change inputs just before next rising edge
      // Values stable before posedge clk
      @(posedge clk);          // DUT samples inputs here
      #1;                      // small delay to ensure sum updated
      $fwrite(outfile, "%0d + %0d = %0d\n", a, b, sum);
    end
  end

  $fclose(infile);
  $fclose(outfile);
  #20 $finish;
end
```

`$fscanf` returns an **integer count** of how many items it successfully read.

Task	Purpose
`$fopen("filename", "r")`	Opens a file for reading
`$feof(file_handle)`	Checks end-of-file
`$fscanf(file_handle, "format", vars)`	Reads formatted data
`$fclose(file_handle)`	Closes the file

What you have now — a “simple” Verilog testbench

Your current testbench:

- Generates clock and reset in separate blocks
- Applies a few inputs manually at `negedge clk`

  Jses `$monitor` to watch outputs

✅ It’s perfect for **functional verification of small RTL** modules — like an adder or counter.

But it’s **flat, manual, and non-reusable.**

Why this doesn’t scale

Imagine testing:

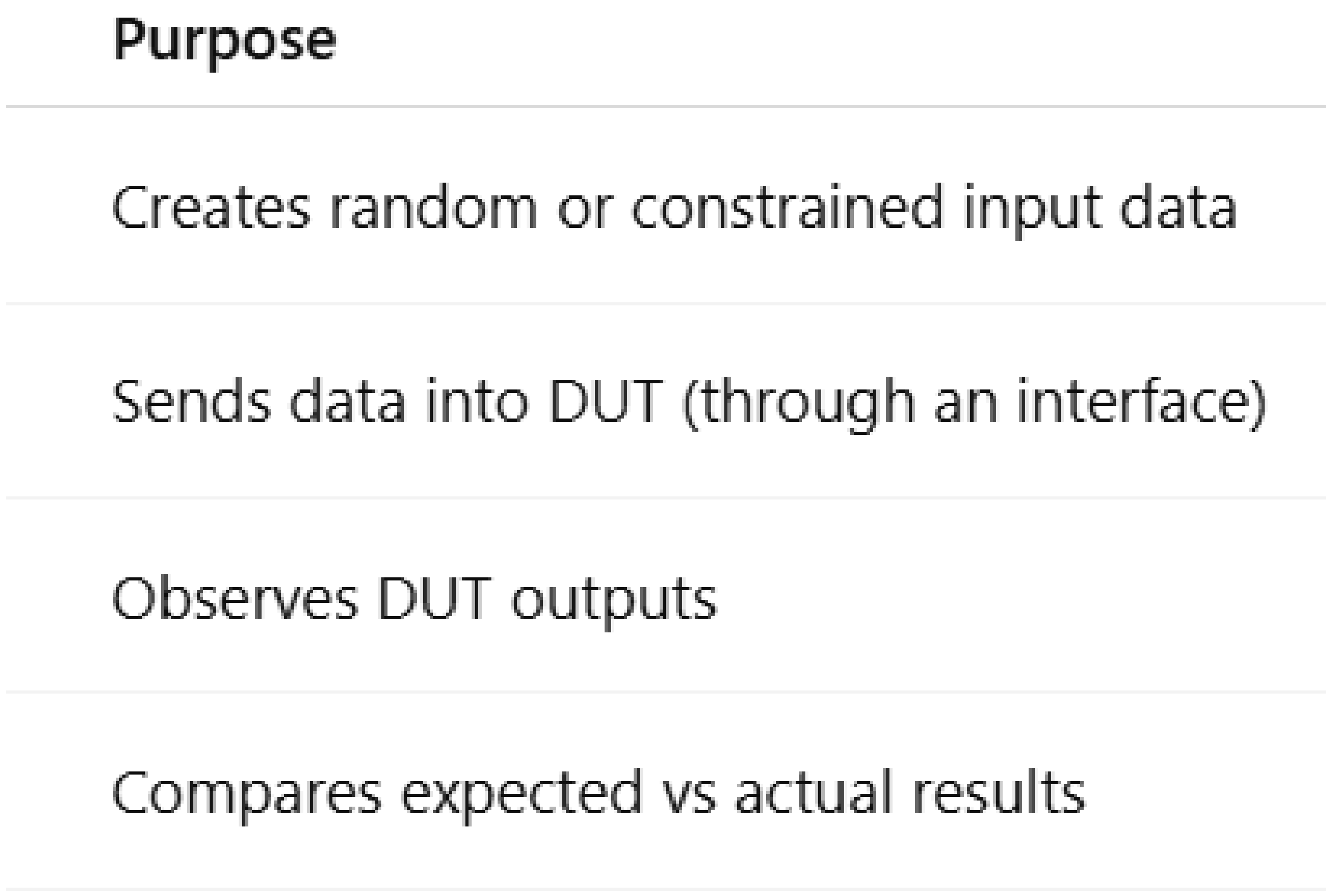
- 32-bit adder
- 1000 random input pairs
- 10 different corner cases (carry, overflow, reset timing, etc.)

You’d need to manually:

- Write loops for random stimulus
- Capture and compare expected outputs
- Add pass/fail checks
- Log results to files

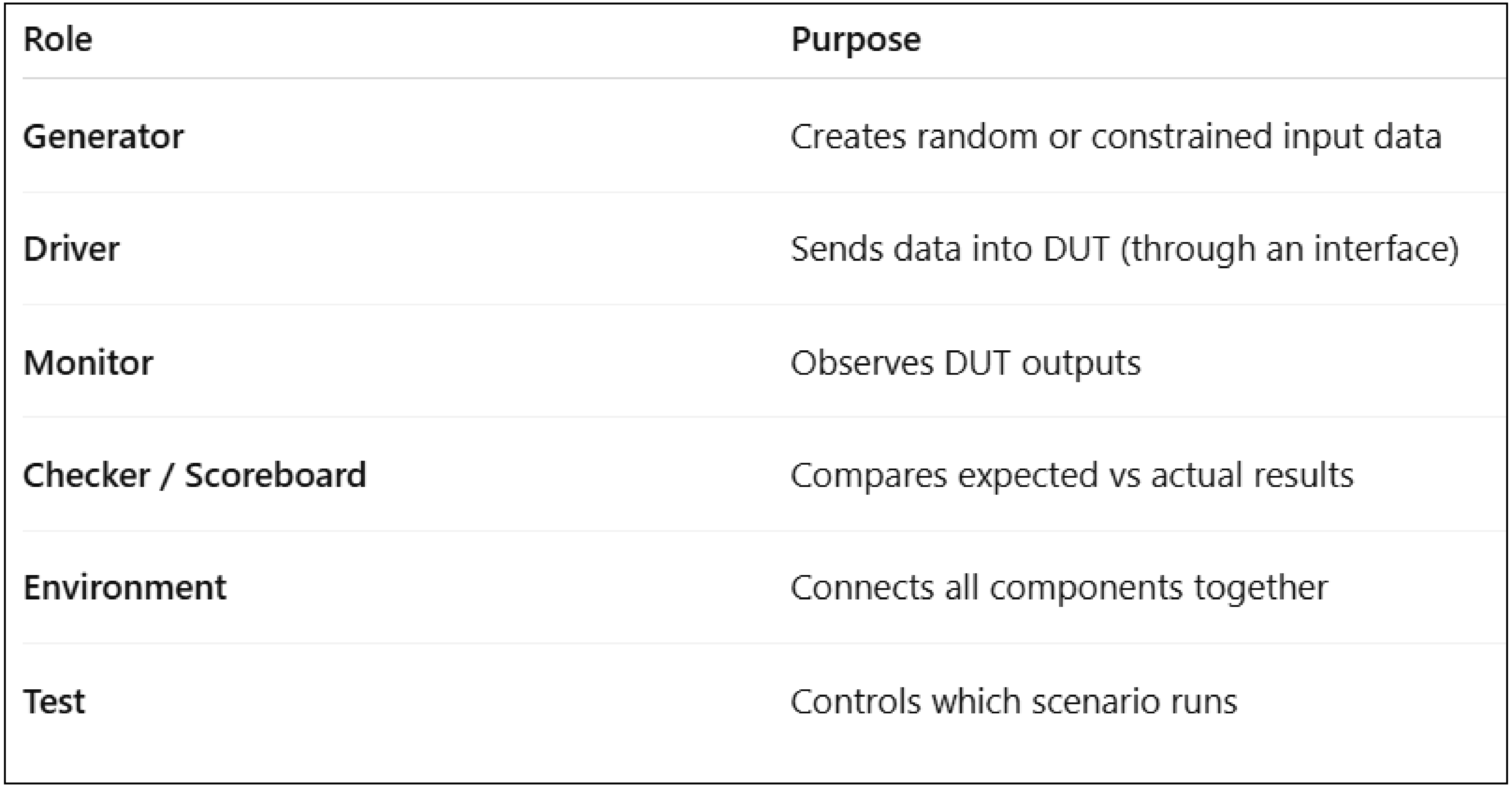
In industry, we call this a **directed testbench** — easy for small DUTs, but **hard to maintain** when design complexity grows.

Structured testbench



Purpose
Creates random or constrained input data
Sends data into DUT (through an interface)
Observes DUT outputs
Compares expected vs actual results

Software framework → Structured TB



Role	Purpose
Generator	Creates random or constrained input data
Driver	Sends data into DUT (through an interface)
Monitor	Observes DUT outputs
Checker / Scoreboard	Compares expected vs actual results
Environment	Connects all components together
Test	Controls which scenario runs

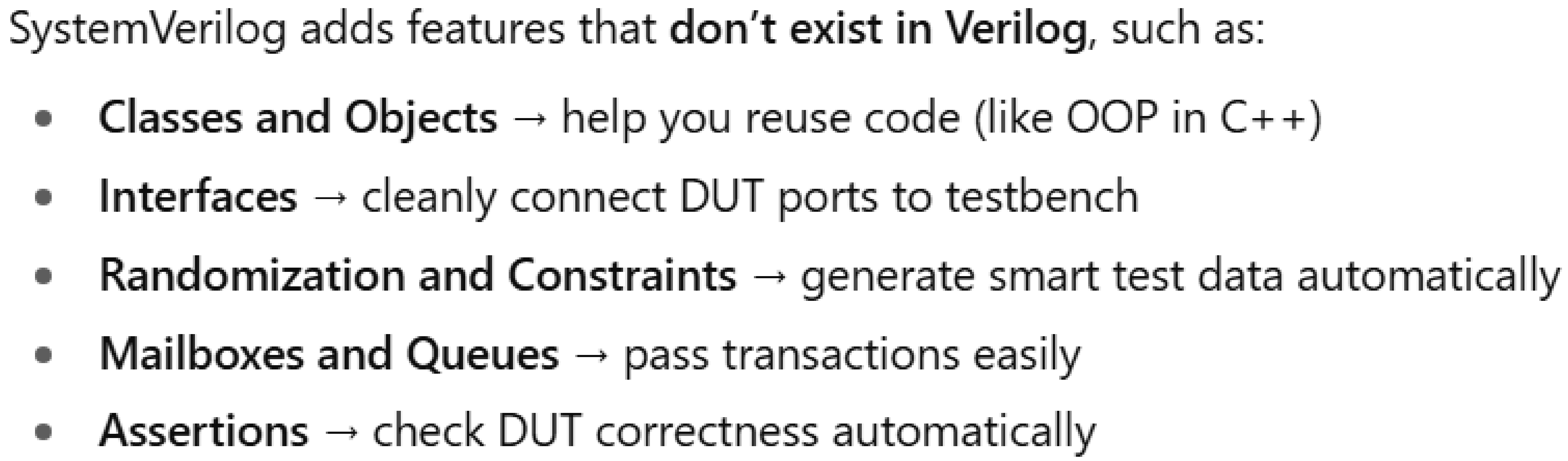
Coding style

- **In plain (directed TB) Verilog, you write all of this in one file;**
- **In structured/Layered TB approach, we split the code into separate blocks**
  - **Classes**
  - **Interface.**

They are reusable.

SystemVerilog

- Modular approach in TB: Reusable classes and interfaces.



SystemVerilog adds features that **don’t exist in Verilog**, such as:

- **Classes and Objects** → help you reuse code (like OOP in C++)
- **Interfaces** → cleanly connect DUT ports to testbench
- **Randomization and Constraints** → generate smart test data automatically
- **Mailboxes and Queues** → pass transactions easily
- **Assertions** → check DUT correctness automatically