SystemVerilog for Functional Verification

What is SystemVerilog?

SystemVerilog is a *hardware description and verification language (HDVL)*

It's an **extension of Verilog** that supports both:

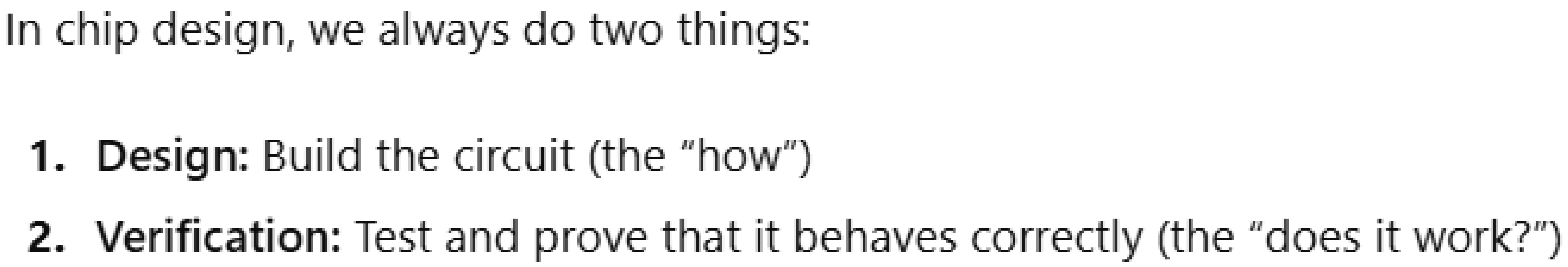
- **Design modelling** (RTL, synthesis, simulation), and
- **Verification**

2002 → SystemVerilog was developed by Accellera

2005 → IEEE 1800-2005: SystemVerilog Standard

What is Verification?

- Verification ensures the RTL design **matches the specification** for *all valid inputs*



In chip design, we always do two things:

1. **Design:** Build the circuit (the "how")
2. **Verification:** Test and prove that it behaves correctly (the "does it work?")

Most important pillars of modern verification:

Randomization → how we *generate test inputs* intelligently

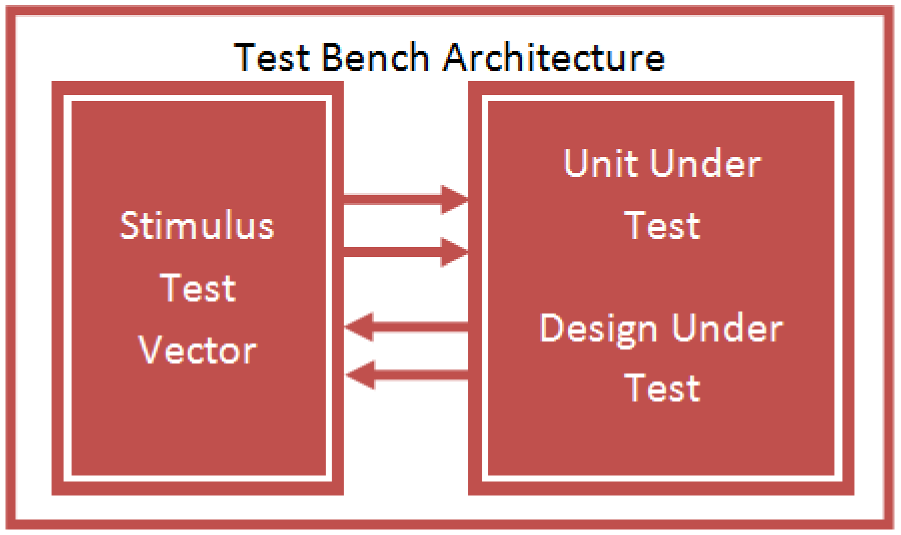
Coverage measurement → how we *measure what we've tested*

Why Randomization is Powerful

- Saves time: you can test thousands of legal scenarios automatically.
- Finds corner cases: unplanned combinations that might cause hidden bugs.

Constrained randomization:

To automatically generate *many different input combinations* that follow certain **rules or constraints**, instead of manually coding every test case



```
a = $random;
b = $random;
```

...but that's **unconstrained** — it gives *any* number (sometimes even invalid).

You cannot tell `$random` things like:

> "a should always be less than b"
> "a and b should not both be zero"
> "select line should be within 0–3 only"

Coverage Measurement

To **measure how much of the design's behaviour,** your tests have actually exercised.

In other words:

"Have I tested *everything* I was supposed to?"

Functional Coverage

Did we test all important scenarios?

Did we try all valid sel values for MUX?

Verilog → Directed

```
// 2x1 Multiplexer
module mux2x1 (
  input  wire a, b,      // data inputs
  input  wire sel,       // select line
  output wire y          // output
);
  assign y = sel ? b : a;
endmodule
```

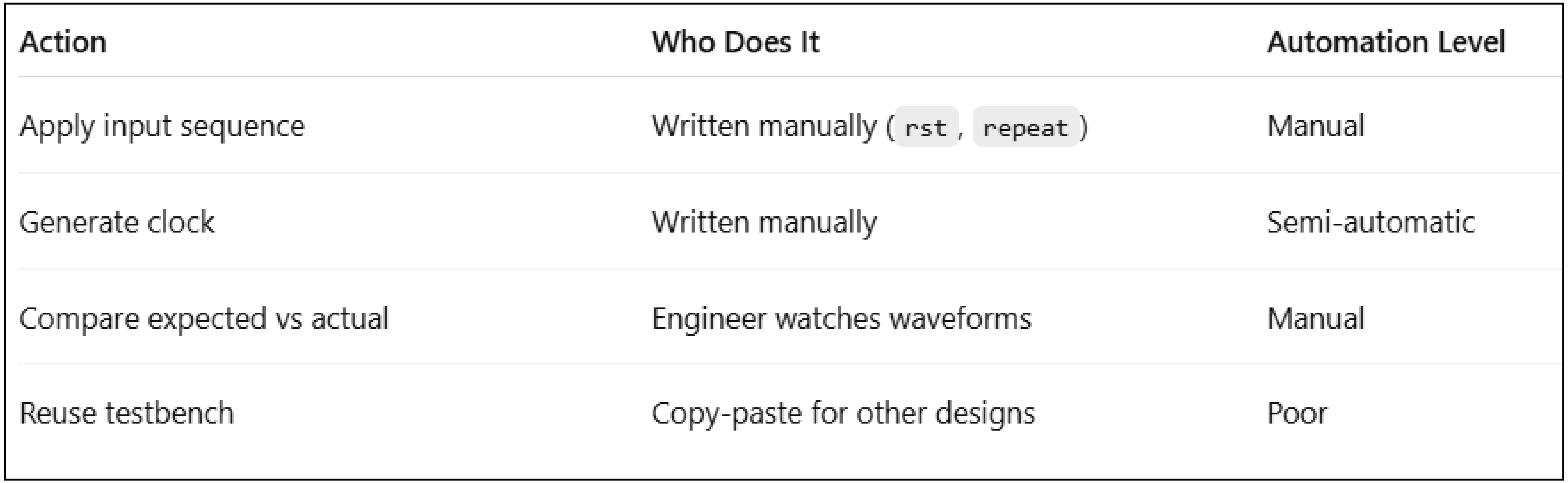
```
`timescale 1ns/1ps
module tb_mux2x1;
  reg a, b, sel;   // inputs
  wire y;          // output

  // Instantiate DUT
  mux2x1 dut (.a(a), .b(b), .sel(sel), .y(y));

  // Stimulus
  initial begin

// Apply test vectors
    a=0; b=0; sel=0; #10; // Expect Y=0
    a=0; b=0; sel=1; #10; // Expect Y=0
    a=0; b=1; sel=0; #10; // Expect Y=0
    a=0; b=1; sel=1; #10; // Expect Y=1
    a=1; b=0; sel=0; #10; // Expect Y=1
    a=1; b=0; sel=1; #10; // Expect Y=0
    a=1; b=1; sel=0; #10; // Expect Y=1
    a=1; b=1; sel=1; #10; // Expect Y=1

    $finish;
  end
endmodule
```

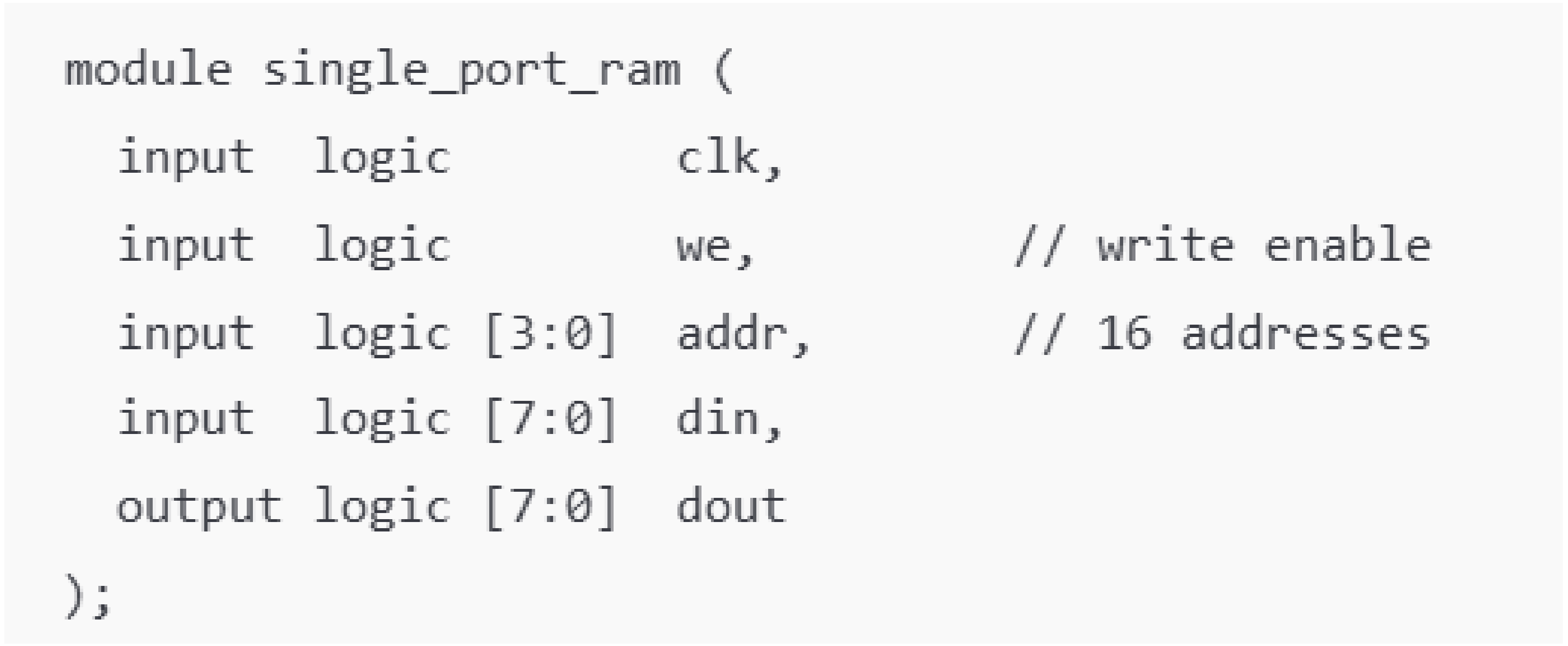


Action	Who Does It	Automation Level
Apply input sequence	Written manually (`rst`, `repeat`)	Manual
Generate clock	Written manually	Semi-automatic
Compare expected vs actual	Engineer watches waveforms	Manual
Reuse testbench	Copy-paste for other designs	Poor

So this works for small modules, but:

- There's **no automatic checking** (no scoreboard)
- **No randomization** of inputs
- **No coverage measurement**

The Design (DUT) — Single-Port RAM



```
module single_port_ram (
  input  logic        clk,
  input  logic        we,         // write enable
  input  logic [3:0]  addr,       // 16 addresses
  input  logic [7:0]  din,
  output logic [7:0]  dout
);
```

What Does "Coverage" Mean Here?

Functional coverage answers:

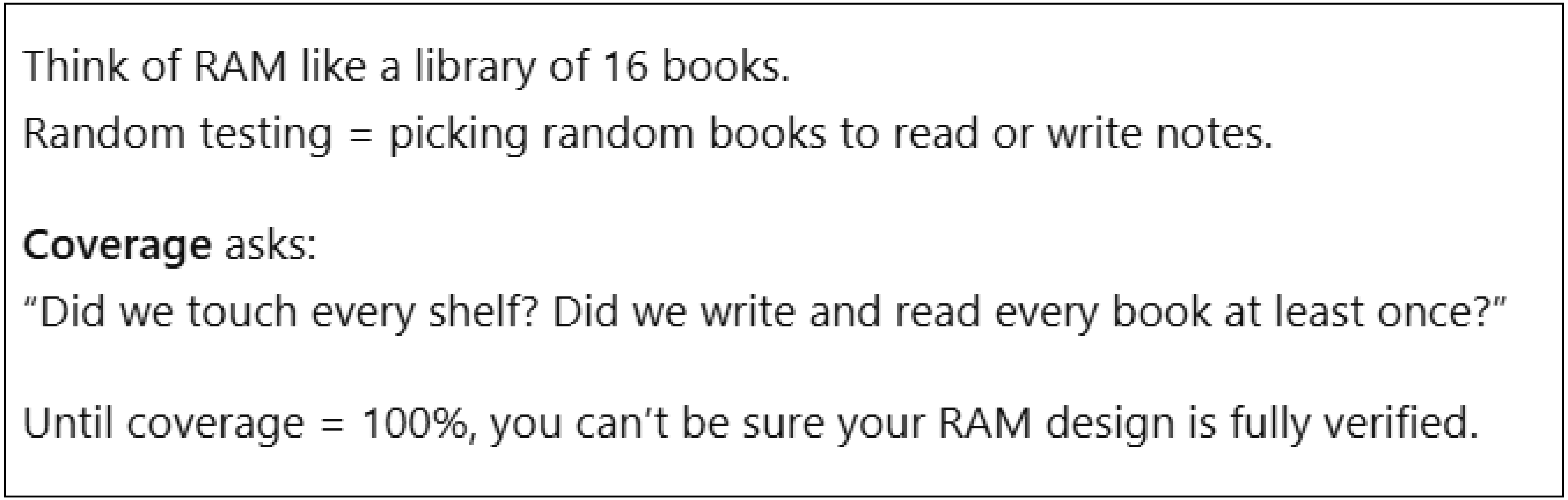
"Have we tested *all possible read/write behaviours* of the RAM?"

That means:

- Have we accessed **all addresses**?
- Have we done both **read and write** operations?
- Have we tested **different data patterns**?
- Have we checked corner cases (e.g., read after write, same address read/write)?

So, functional coverage helps measure **how thoroughly we've exercised the RAM's functionality**, not just how many times it ran.

Analogy

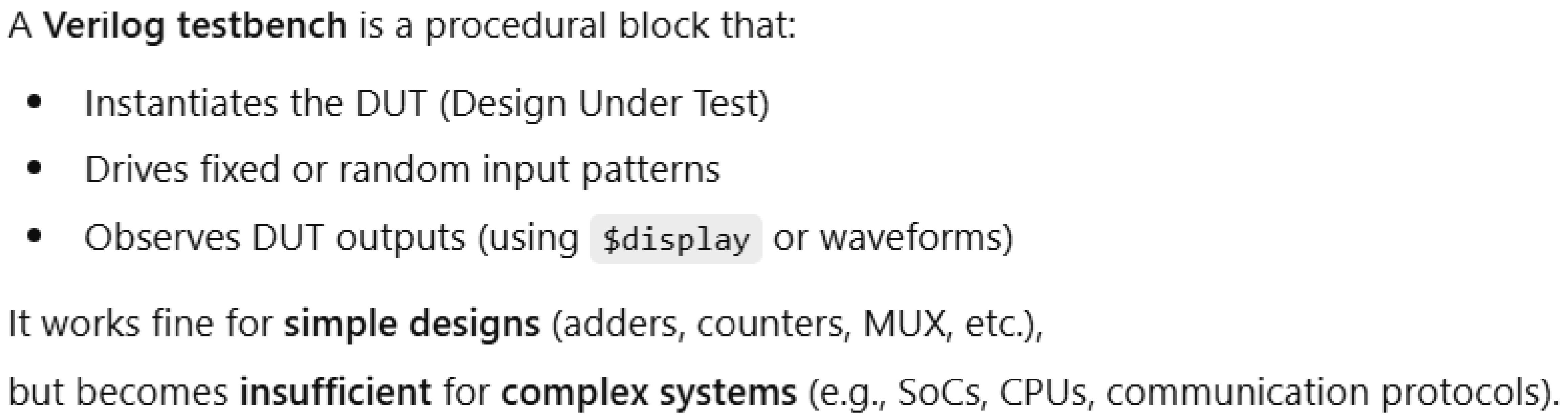


Think of RAM like a library of 16 books.
Random testing = picking random books to read or write notes.

Coverage asks:
“Did we touch every shelf? Did we write and read every book at least once?”

Until coverage = 100%, you can’t be sure your RAM design is fully verified.

A Verilog testbench is like a manual tester who runs fixed tests and looks at results on paper.



A **Verilog testbench** is a procedural block that:

- Instantiates the DUT (Design Under Test)
- Drives fixed or random input patterns
- Observes DUT outputs (using `$display` or waveforms)

It works fine for **simple designs** (adders, counters, MUX, etc.),

but becomes **insufficient** for **complex systems** (e.g., SoCs, CPUs, communication protocols).

What we need…

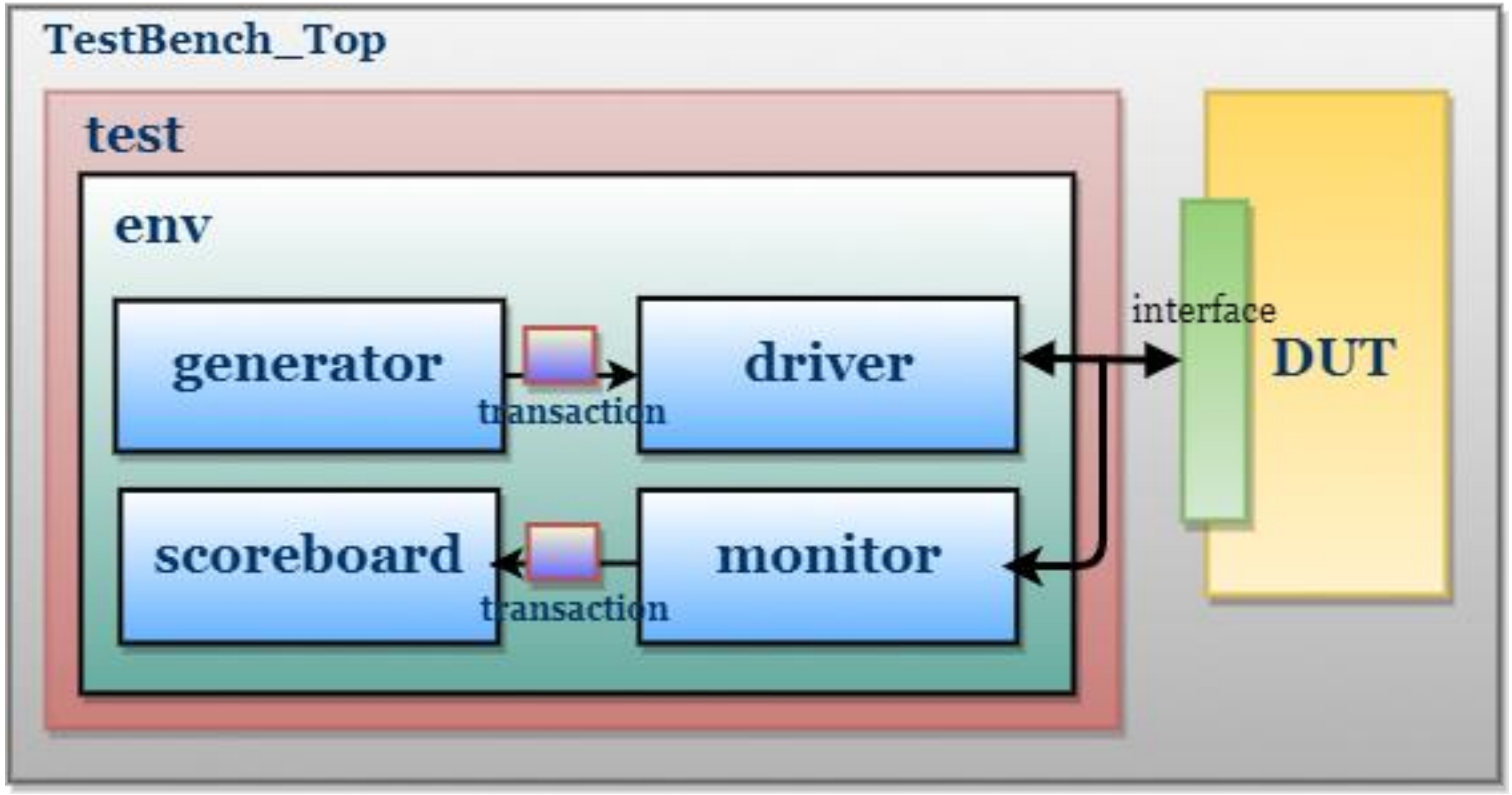
A software framework which creates an ***intelligent testing environment*** that can **generate thousands of valid scenarios, check correctness automatically, and measure what's been tested (coverage).**

Why SV?

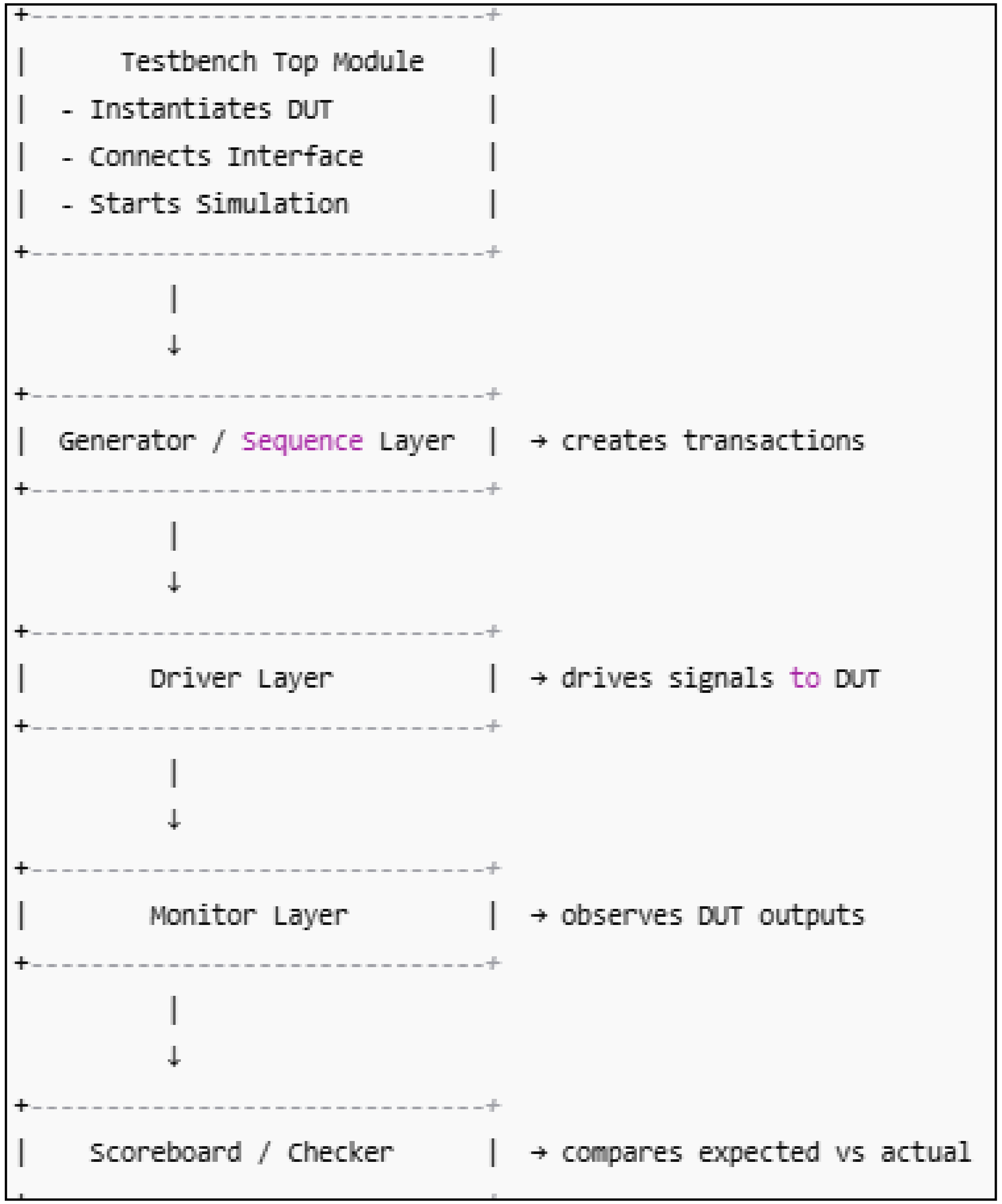
- You *can* create your own layered testbench using SystemVerilog
- A **layered testbench** organizes verification logic into modular layers — so that stimulus generation, driving, monitoring, checking, and coverage are separated.
- You can use **UVM** which is a framework built on top of SystemVerilog — it defines a standardized structure for building reusable verification environments.

The word **“layered”** simply means that the testbench is **organized in distinct levels (or layers)** —
each layer performs one **specific function** in the verification process.

A layered testbench has multiple logical layers that work together to verify the DUT (Device Under Test).



Layer	Purpose
Generator Layer	Creates or randomizes test data
Driver Layer	Sends that data to the DUT inputs
DUT (Device Under Test)	The design you're verifying (e.g., MUX)
Monitor Layer	Watches what DUT outputs
Scoreboard Layer	Compares DUT output vs. expected result



```
+-------------------------------+
|      Testbench Top Module     |
|  - Instantiates DUT           |
|  - Connects Interface         |
|  - Starts Simulation          |
+-------------------------------+
          |
          ↓
+-------------------------------+
|  Generator / Sequence Layer   |  → creates transactions
+-------------------------------+
          |
          ↓
+-------------------------------+
|        Driver Layer           |  → drives signals to DUT
+-------------------------------+
          |
          ↓
+-------------------------------+
|        Monitor Layer          |  → observes DUT outputs
+-------------------------------+
          |
          ↓
+-------------------------------+
|    Scoreboard / Checker       |  → compares expected vs actual
```

Verification framework

- The industry needed a **common verification framework** to avoid each company writing its own structure.
- The **Universal Verification Methodology (UVM)** was born — built entirely *on top of SystemVerilog* using its OOP features.
- UVM is not a separate language — it’s a SystemVerilog class library.

UVM (Universal Verification Methodology)

A library/framework written in SystemVerilog (SV) that defines how to organize and reuse features (randomization, coverage, OOP, interfaces, and assertions) of verification setup.

"SV is the engine — UVM is the car built around it."

SV datatype

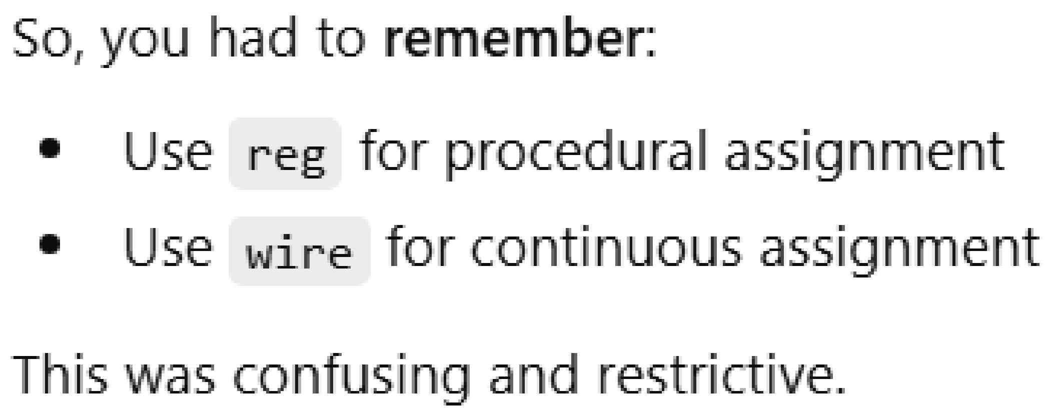
In Verilog, data types were simple but limited — mostly just reg, wire, and integer.

SystemVerilog extended this to provide **stronger typing, 4-state logic control, signed arithmetic, and high-level verification types**

Verilog rule --> “Net type wire cannot be assigned in a procedural block.”

```
module and_gate1;
  reg a, b, y;
  always @(*) begin
    y = a & b;    // Procedural assignment
  end
endmodule
```

```
module and_gate2;
  wire a, b, y;
  assign y = a & b;  // Continuous assignment
endmodule
```



So, you had to **remember**:

- Use `reg` for procedural assignment
- Use `wire` for continuous assignment

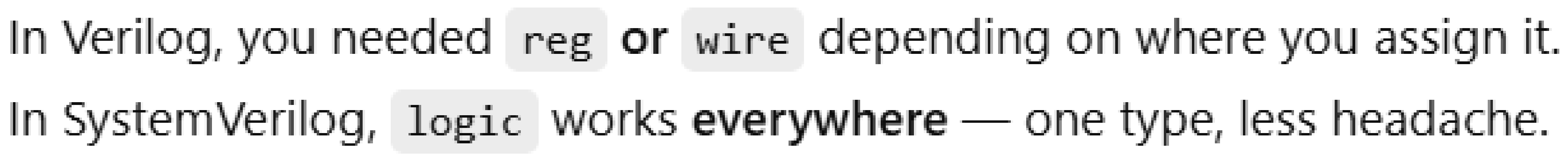
This was confusing and restrictive.

SystemVerilog: Use logic for both!

```
module and_gate1;
  logic y;
  always @(*) begin
    y = a & b;    // Procedural assignment
  end
endmodule
```

```
module and_gate2;
  logic a, b, y;
  assign y = a & b;  // Continuous assignment only
endmodule
```

logic can be used in both continuous and procedural assignments.



In Verilog, you needed `reg` or `wire` depending on where you assign it.

In SystemVerilog, `logic` works **everywhere** — one type, less headache.

SystemVerilog was born as an *extension* of Verilog

- SystemVerilog wasn’t created to replace Verilog, but to **extend** it.
- Its early versions (IEEE 1800-2005) were meant mainly for **verification**, not RTL.
- So, for **RTL module design**, Verilog’s limited feature set is **enough and safer**. SystemVerilog adds complexity that’s often *not needed* for pure RTL.

Legacy Code and IP Reuse

- The semiconductor industry has **millions of lines of legacy Verilog RTL**.
- Companies rarely rewrite working IP in SV just for style.
- Verification teams may use SV/UVM around the same RTL —
  but the RTL itself remains Verilog for **stability and compatibility.**

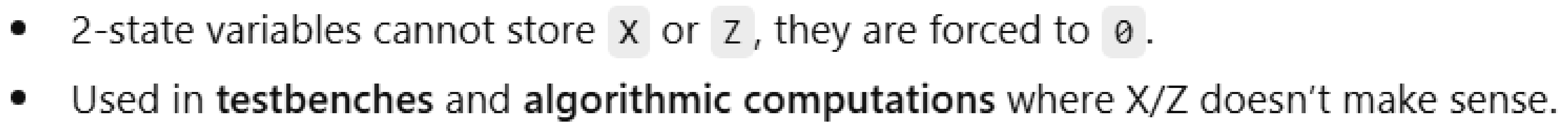
SystemVerilog defines **how many logic values** a variable can represent.

Type	Values	Meaning
2-state	`0`, `1`	Only binary values
4-state	`0`, `1`, `X`, `Z`	Includes unknown (`X`) and high-impedance (`Z`)

```
bit [3:0] a;     // 2-state variable

a = 4'b10xz;     // X and Z converted to 0 automatically

$display("a = %b", a); // prints a = 1000
```



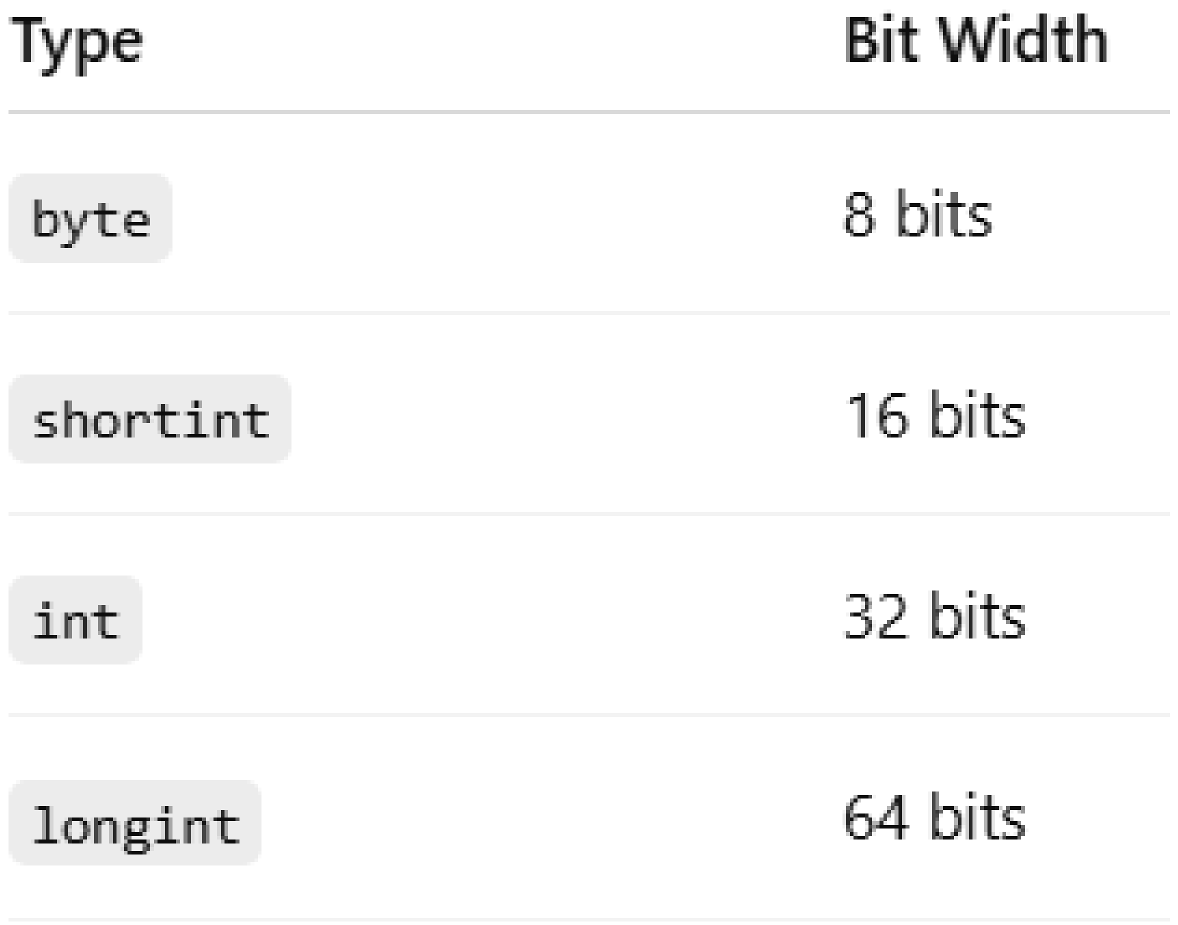
- 2-state variables cannot store `X` or `Z`, they are forced to `0`.
- Used in **testbenches** and **algorithmic computations** where X/Z doesn't make sense.

```
logic [3:0] b;

b = 4'b10xz;

$display("b = %b", b); // prints b = 10XZ
```

SystemVerilog introduced **four standardized, sized integer types**

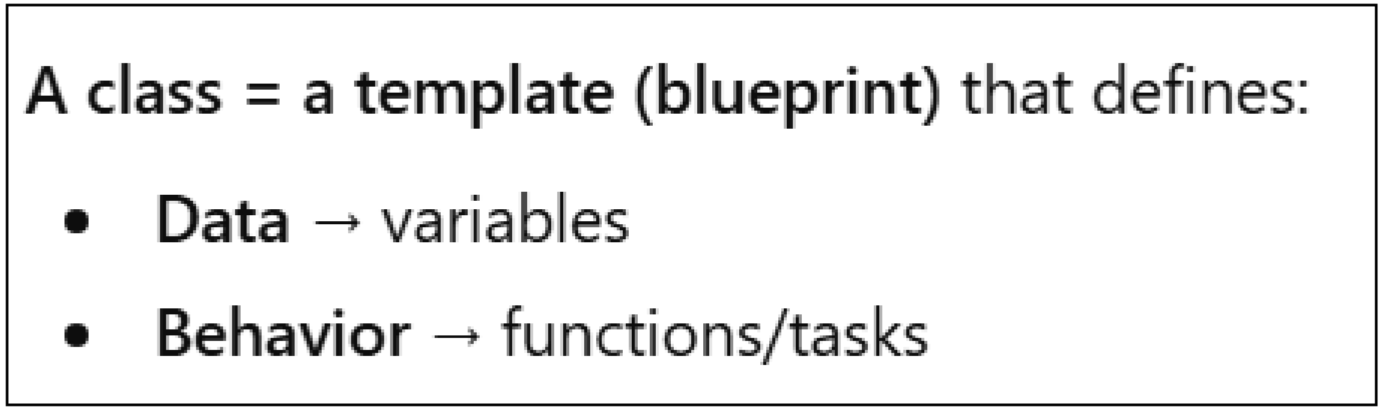


Type	Bit Width
`byte`	8 bits
`shortint`	16 bits
`int`	32 bits
`longint`	64 bits

OOP in SystemVerilog

For big verification projects, we need to **group related data and functions** together and reuse them easily.

Introduce the Concept of a Class



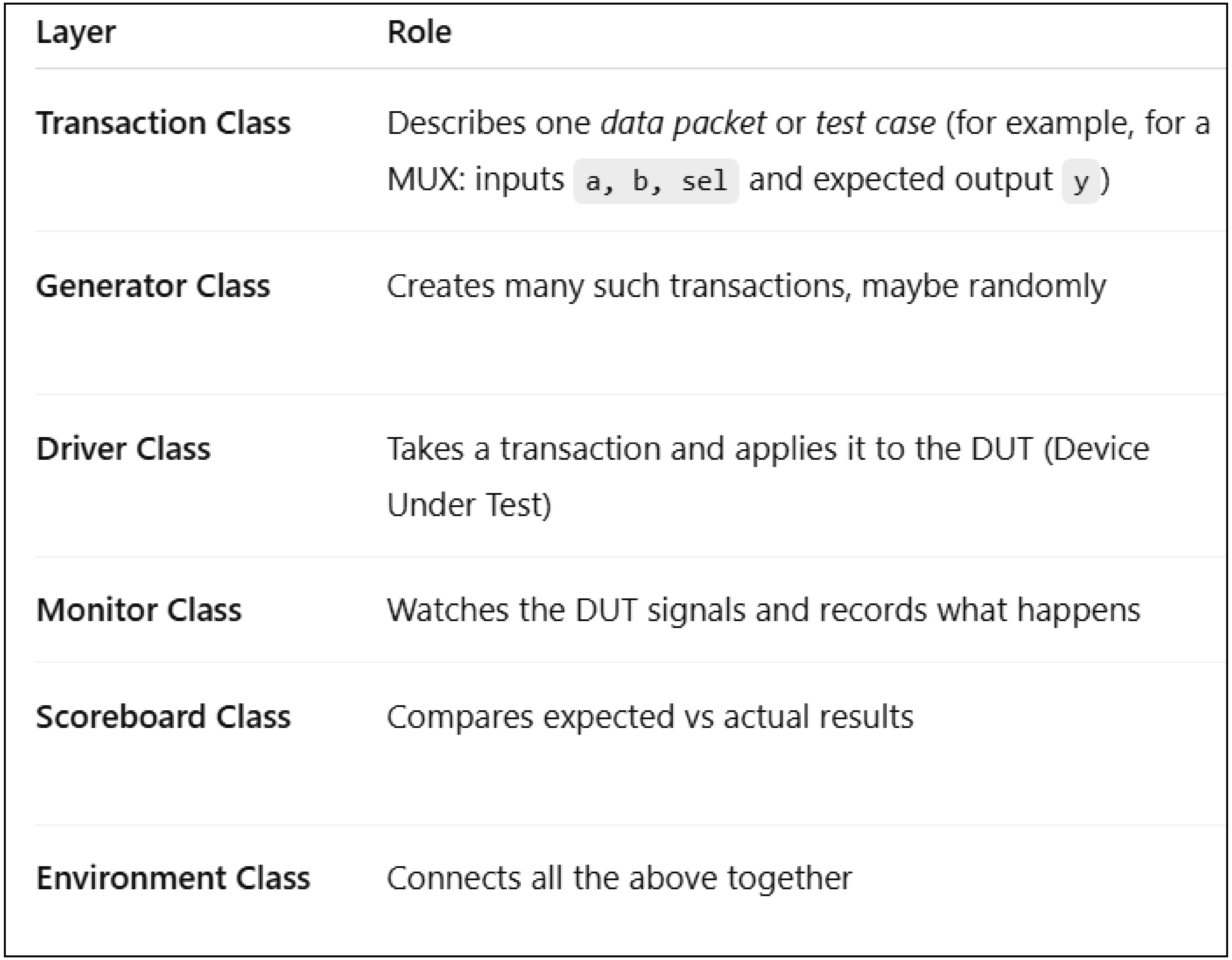
A class = a template (blueprint) that defines:

- **Data** → variables
- **Behavior** → functions/tasks

Concept	What It Shows
Inheritance	Reuse and extend parent class
super.new()	Calls parent constructor
Method Overriding	Redefine a function's behavior
Encapsulation	Data + functions grouped in one class
Objects	Individual student records

A layered testbench is an OOP-based verification structure where each class (generator, driver, monitor, scoreboard) plays a specific role — working together through objects instead of procedural code.

That’s the foundation of a layered testbench —
and the same idea scales up to UVM.



Layer	Role
Transaction Class	Describes one *data packet* or *test case* (for example, for a MUX: inputs `a, b, sel` and expected output `y`)
Generator Class	Creates many such transactions, maybe randomly
Driver Class	Takes a transaction and applies it to the DUT (Device Under Test)
Monitor Class	Watches the DUT signals and records what happens
Scoreboard Class	Compares expected vs actual results
Environment Class	Connects all the above together

Inheritance and Constructors in SystemVerilog

- Inheritance allows child classes to reuse and extend base classes.
- Supports method overriding using virtual/override.
- Promotes modular and reusable testbench components.
- Constructors (new) allocate memory and initialize objects.

Role of Constructor (new)

- Object handles are null until new is called.
- new creates the actual object instance.
- Can include arguments for initialization.
- Essential for safe access of class members.

Classes, Objects & Mailboxes in Testbench

- Classes define generator, driver, monitor, scoreboard.
- Objects are runtime instances enabling modularity.
- Mailboxes provide thread-safe communication.
- Enable transaction flow between testbench layers.

Mailbox-Based Communication Flow

- Generator → sends transactions → Driver
- Driver → drives DUT
- Monitor → captures DUT outputs → sends to Scoreboard
- Scoreboard → compares expected vs actual behavior