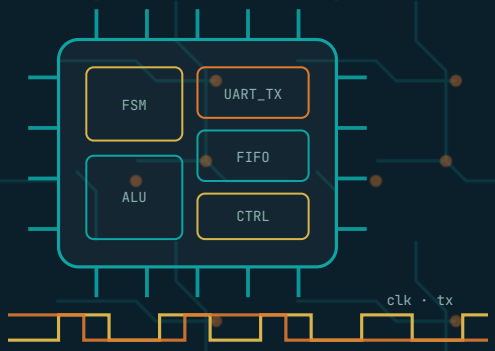
STUDY PAGE · AREA 02 OF 14

Verilog HDL & RTL Basics

02

How Verilog describes hardware, how the simulator schedules it, and the first testbench that checks a real design — a UART transmitter.



16 h of study | **4** modules | **9** diagrams · **3** interactive | **Free** tools · Windows, macOS, Linux | **20**-question test · 70% to pass | Lab: UART TX + testbench

1 · Modules — ports · nets · 4-state
→ 2 · = vs <= — event queue · races
→ 3 · FSMs — two-process style
→ 4 · Testbench — driver · monitor · check
→ Lab · UART — find the planted bug
→ Area test · 70% — 20 questions

BEFORE YOU START · A

Who this area is for

Verilog is the language every chip, FPGA and verification job is built on. If any of these describe you, start here.



EE / ECE students

Final-year and master's students preparing for VLSI internships and campus placements in design or verification.

Software engineers

Developers moving into hardware: you already know loops and functions; this area teaches you why hardware code runs *in parallel*.

QA & test engineers

Test automation people heading into design verification — the same mindset (stimulus, check, coverage) applied to silicon.

FPGA & embedded engineers

You write some Verilog already and want the simulator's rules, race-free coding and a proper testbench habit.

RTL designers

Designers who want to move to verification and need to think like the person checking their code.

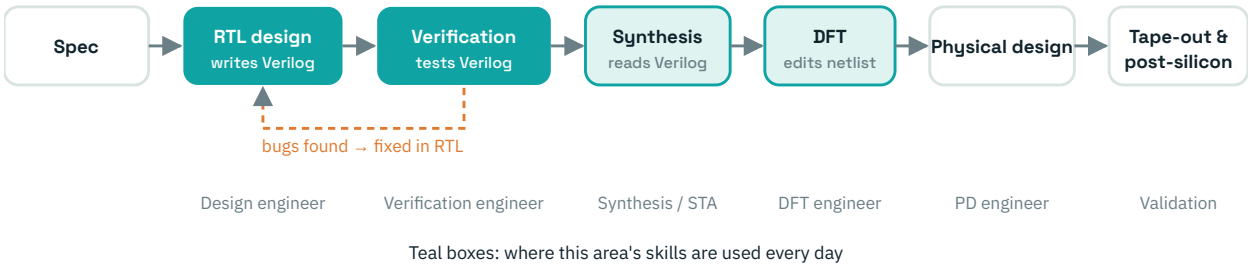
Career returners

Engineers refreshing HDL fundamentals before interviews after time away from hands-on work.

Prerequisites

You need	Level	Where to get it
Binary, hex, two's complement	Comfortable	Area 01 · Module 1
Gates, multiplexers, adders	Can draw them	Area 01 · Module 3
Flip-flops, setup/hold, state machines	Know the idea	Area 01 · Module 4
Any programming language	Basic (variables, if, loops)	Helpful, not required
Command line	Can run a command and read its output	Install guide below walks you through it

Where Verilog sits in the chip flow — and the roles that use it



Teal boxes: where this area's skills are used every day

Verilog is the hand-off format of the whole flow. Designers write it, verification engineers test it, synthesis turns it into gates and DFT engineers edit the result.

Role	How you use this area on the job	Typical first title
Design verification (DV)	Read RTL to debug failures, write testbenches, understand races in waveforms	DV Engineer / Verification Engineer I
RTL design	Write synthesizable modules, FSMs and clean resets	ASIC Design Engineer / RTL Engineer
FPGA	Write and simulate Verilog before loading it on a board	FPGA Engineer
DFT	Read netlists and RTL, simulate scan and BIST patterns	DFT Engineer
Post-silicon validation	Reproduce silicon failures in simulation, read the RTL behind them	Validation Engineer

BEFORE YOU START · B

What you will be able to do

Every outcome is practised in the lab and checked by the area test.



- Write modules with ports, parameters, nets and variables, and connect them by name.
- Explain how `assign`, `always @*` and `always @(posedge clk)` become hardware.
- Predict blocking and non-blocking results and spot race conditions in a waveform.
- Describe the simulator's event regions and why `$display` and `$strobe` differ.
- Write a synthesizable two-process state machine with a clean reset and no latches.
- Tell synthesizable code from simulation-only code.

- [x] Build a clocked testbench with a driver, monitor, expected queue and checker.
- [x] Find a real bug in a UART transmitter from the first mismatch.

BEFORE YOU START · C

Tools & software

Everything in this area runs on free, open-source tools. Commercial simulators used at work are listed so you recognise them in job posts.



Tool	What it does	Used in this area for	Cost
Icarus Verilog `iverilog` + `vvp`	Event-driven Verilog simulator: compiles then runs	Every example and the lab (`make sim`)	Free
Verilator	Very fast simulator that compiles Verilog/SystemVerilog to C++; also a strict linter	Second simulator and lint checks (`make verilator`)	Free
GTKWave or **Surfer**	Waveform viewers for `.vcd` / `.fst` dumps	Looking at clocks, states and the UART line	Free
VS Code + Verilog extension	Editor with syntax highlighting and lint on save	Writing RTL and testbenches	Free
EDA Playground	Browser-based simulators, including commercial ones, after a free sign-up	Zero-install option; also lets you try commercial tools	Free
make and **git**	Run builds with one command; track your changes	Lab Makefile; saving your work	Free

Commercial simulators you will meet at work

Simulator	Vendor	Notes
VCS	Synopsys	Widely used for SystemVerilog/UVM regressions
Xcelium	Cadence	Successor of Incisive / NC-Sim; often paired with SimVision or Verisium
Questa	Siemens EDA	Successor of ModelSim; ModelSim/Questa starter editions ship with FPGA tools

i

Same language, same rules

Code that is correct by the standard (IEEE 1364 Verilog, IEEE 1800 SystemVerilog) behaves the same on free and commercial simulators. Learning on Icarus and Verilator transfers directly; only the command lines change.

BEFORE YOU START · D

Install step by step

Pick your computer. About 15 minutes. If anything fails, the browser option works with nothing installed.



Windows

Recommended: WSL 2 (Ubuntu inside Windows). You get the same commands as Linux and as most companies' compute farms.

1

Turn on WSL and install Ubuntu

Open **PowerShell as Administrator** and run the command below, then restart the computer.

```
wsl --install
```

After the restart an Ubuntu window opens: choose a user name and password.

2 **Update Ubuntu and install the tools**

In the Ubuntu window:

```
sudo apt update
sudo apt install -y iverilog verilator gtkwave make git
```

3 **Install VS Code and connect it to WSL**

Install VS Code from code.visualstudio.com, add the **WSL** extension, then in Ubuntu type `code .` to open the current folder.

4 **Add Verilog support**

In VS Code's Extensions panel install **Verilog-HDL/SystemVerilog/Bluespec SystemVerilog** (publisher mshr-h).

5 **Check it works**

```
iverilog -V | head -1
verilator --version
```

Each prints a version line. Waveforms: run `gtkwave` (Windows 11 shows Linux windows directly).

i **No WSL allowed on your laptop?**

Install the native Icarus Verilog for Windows build (installer from bleyer.org/icarus, which also bundles GTKWave) and tick "Add executable folder to the user PATH". Use EDA Playground for Verilator.

macOS

1 **Install Apple's command-line tools**

```
xcode-select --install
```

2 **Install Homebrew**

Follow the one-line command on brew.sh, then close and reopen Terminal.

3 **Install the simulators**

```
brew install icarus-verilog verilator make git
```

4 **Install a waveform viewer**

Download **Surfer** from surfer-project.org (runs natively on Apple silicon), or try `brew install --cask gtkwave`.

5 **Editor**

Install VS Code and the **Verilog-HDL/SystemVerilog/Bluespec SystemVerilog** extension.

6 **Check it works**

```
iverilog -V | head -1
verilator --version
```

Linux

1 **Ubuntu / Debian**

```
sudo apt update
sudo apt install -y iverilog verilator gtkwave make git
```

2 **Fedora**

```
sudo dnf install -y iverilog verilator gtkwave make git
```

3 **Arch**

```
sudo pacman -S iverilog verilator gtkwave make git
```

4 **Check the versions**

```
iverilog -V | head -1      # 11 or newer
verilator --version        # 5.x recommended for later areas
```

If your distribution ships Verilator 4.x, later SystemVerilog areas need 5.x: build it from source following verilator.org, or use EDA Playground.

Browser only

1 **Open EDA Playground**

Go to edaplayground.com and create a free account (simulation needs a login).

2 **Paste the code**

Design file on the right, testbench on the left.

3 **Choose a simulator**

Under *Tools & Simulators* pick **Icarus Verilog** for this area; tick *Open EPWave after run* to see waveforms.

4 **Run**

Press **Run**. The log at the bottom shows your `$display` output.

! **Waveforms in the browser**

EPWave needs a dump in your testbench: `initial begin $dumpfile("dump.vcd"); $dumpvars; end`.

BEFORE YOU START · E

Your first simulation in 5 minutes

Prove the tools work end to end: write, compile, run, look at a waveform.



1 **Make a folder and a file called `hello.v`**

```
module hello;
  reg clk = 0;
  always #5 clk = ~clk;                // 10 ns clock
  initial begin
    $dumpfile("hello.vcd"); $dumpvars(0, hello);
    $display("Hello, hardware! t=%0t", $time);
    #50 $finish;
  end
endmodule
```

2 **Compile it**

```
iverilog -g2012 -o hello.vvp hello.v
```

`-g2012` turns on SystemVerilog-2012 features; `-o` names the compiled output.

3 **Run it**

```
vvp hello.vvp
```

Expected output:

```
VCD info: dumpfile hello.vcd opened for output.
Hello, hardware! t=0
hello.v:7: $finish called at 50 (1s)
```

4

Open the waveform

```
gtkwave hello.vcd
```

Click `hello` in the hierarchy, drag `clk` into the signal list: five full clock periods.

BEFORE YOU START · F

Two-week study plan

16 hours at about 8 hours a week. Read, try every code block, then lab, then test.



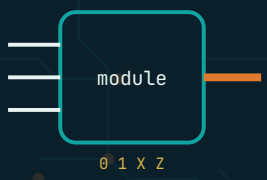
Session	Do this	Time
Day 1	Install tools · first simulation · Module 1 (modules, nets, four-state, vectors)	3 h
Day 2	Module 2 (hardware mapping, blocking vs non-blocking, event queue); try the shift-register switch	3 h
Day 3	Module 3 (FSMs, synthesizable subset); type the UART FSM and simulate it	2.5 h
Day 4	Module 4 (testbench architecture, UART monitor)	2.5 h
Day 5	Lab: complete the monitor, find and fix the bug, reach PASS	3 h
Day 6	Cheat sheet review · interview questions · area test (70% to pass)	2 h

MODULE 1 OF 4

Modules, ports & nets

A module is a box of hardware. Learn to read its boundary, what flows across it, and which values a wire can hold.

- Module anatomy: ports, parameters, instances
- Nets vs variables, and why `reg` is misleading
- Four-state logic: 0, 1, X, Z
- Vectors, part-selects, concatenation



⏱ 4 hours · Area 02 · Verilog

Start from zero

New to this? Work through these steps first, in order. Already comfortable? Skim them and jump to the reference below.

1 STEP 1 — Verilog describes hardware, not steps

≈ In plain words

A software program runs one line after another. Verilog is different: each statement describes a **piece of hardware that exists all the time**, and all the pieces work at once.

✳ Think of it like

A recipe says “first do this, then that”. A building plan says “here is a door, here is a window” — they all exist together. Verilog is the building plan.

✎ Worked example

```
assign y = a & b;   // not “compute once”: an AND gate tha
```

Whenever `a` or `b` changes, `y` changes — forever, just like a real ga

Aa The word for it

A **hardware description language (HDL)**. The code you verify is called **RTL** (register-transfer level).

Why it matters

If you read RTL like a program you will predict the wrong values. Read it as a circuit: every line is always active.

STEP 2

2 A module is a box with pins

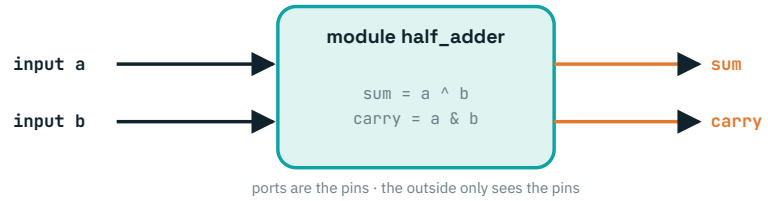
In plain words

Every design is built from **modules**. A module is a box: it has named pins (ports) for signals going in and out, and logic inside.

Think of it like

A phone charger: you only care about the socket and the plug. What's inside is its own business.

Worked example



ports are the pins · the outside only sees the pins

The half adder from Area 01, as a Verilog module.

```
module half_adder (input  a, b,
                   output sum, carry);
  assign sum   = a ^ b;
  assign carry = a & b;
endmodule
```

The word for it

module, **ports** (`input`, `output`, `inout`).

Why it matters

The design you verify is a module (the **DUT**, device under test). Your testbench only touches its ports.

STEP 3

3 Modules inside modules

≈ **In plain words**

Big designs are built by placing modules inside other modules and wiring their ports together.

☀ **Think of it like**

LEGO: small bricks make a wall, walls make a house.

✎ **Worked example**

```
module full_adder (input a, b, cin, output sum, cout);
  wire s1, c1, c2;
  half_adder ha1 (.a(a),  .b(b),   .sum(s1),  .carry(c1));
  half_adder ha2 (.a(s1), .b(cin), .sum(sum), .carry(c2));
  assign cout = c1 | c2;
endmodule
```

`.a(s1)` means: plug my wire `s1` into the copy's pin `a`.

Aa **The word for it**

Each placed copy is an **instance**; the result is a **hierarchy**.

! **Why it matters**

Waveform viewers show the hierarchy as a tree (`tb.dut.ha2.sum`). You will navigate it daily to find signals.

4 STEP 4 wire versus reg

≈ **In plain words**

A `wire` just shows whatever drives it, all the time. A `reg` holds the last value an `always` block gave it.

☀ **Think of it like**

A wire is a water pipe: it carries whatever flows in. A reg is a bucket: it keeps what you last poured in until you pour again.

✎ **Worked example**

You write	Use	Becomes
`assign y = a | b;`	`wire y`	a gate
`always @(*) y = a | b;`	`reg y`	the same gate!
`always @(posedge clk) q <= d;`	`reg q`	a flip-flop

Aa **The word for it**

Nets (wire) and **variables** (reg). Despite the name, a `reg` is **not** always a register.

! **Why it matters**

Seeing `reg` and assuming “flip-flop” is a classic beginner mistake when reading RTL.

5

STEP 5

Every bit has four values: 0, 1, X, Z

≈ **In plain words**

In simulation a bit can be 0, 1, **X** (unknown) or **Z** (not driven by anyone).

☀ **Think of it like**

X is a smudged meter reading — it's there but you can't tell what. Z is an unplugged cable — nothing is driving it.

✎ **Worked example**

Situation	Value
reg never assigned, or no reset	X
wire with nothing connected	Z
two drivers fighting (0 and 1)	X
`X & 0`	0 (AND with 0 is 0 anyway)
`X & 1`	X

Aa **The word for it** — **Four-state logic**.

! **Why it matters** — Real silicon has no X — so an X in simulation is a warning that hardware may behave randomly. Good checkers treat X as a failure.

PAUSE & TRY **What is `1'bx | 1'b1` ? And `1'bx | 1'b0` ?** Answer

OR with 1 is always 1 → **1**. OR with 0 passes the other input → **X**.

6 STEP 6

Vectors: many bits in one name

≈ **In plain words** — A vector is a group of bits with one name, like an 8-bit bus. You can pick single bits, slices, join vectors and repeat them.

☀ **Think of it like** — A ribbon cable of 8 wires, numbered 7 down to 0.

Worked example

1. `wire [7:0] data = 8'hB6;` → `1011_0110`.
2. One bit: `data[7]` = 1.
3. A slice: `data[3:0]` = `4'h6`.
4. Join (concatenate): `{data[3:0], data[7:4]}` = `8'h6B` (nibbles swapped).
5. Repeat: `{4{1'b1}}` = `4'b1111`.

The word for it

Vector, **part-select**, **concatenation** `{ }`, **replication** `{n{ }}`.

Why it matters

Packets, registers and buses are all vectors; your checks slice them into fields.

PAUSE & TRY **What is `{2'b10, 2'b01}` ? And `8'hB6` bits `[7:4]` ?** Answer

`4'b1001`. Bits [7:4] of `1011_0110` are `1011` = **4'hB**.

STEP 7

7 Number literals: size, base, value

In plain words

Verilog numbers are written as **size ' base value**.

Think of it like

Like writing a price with its currency: “8 bits, hex, B6”.

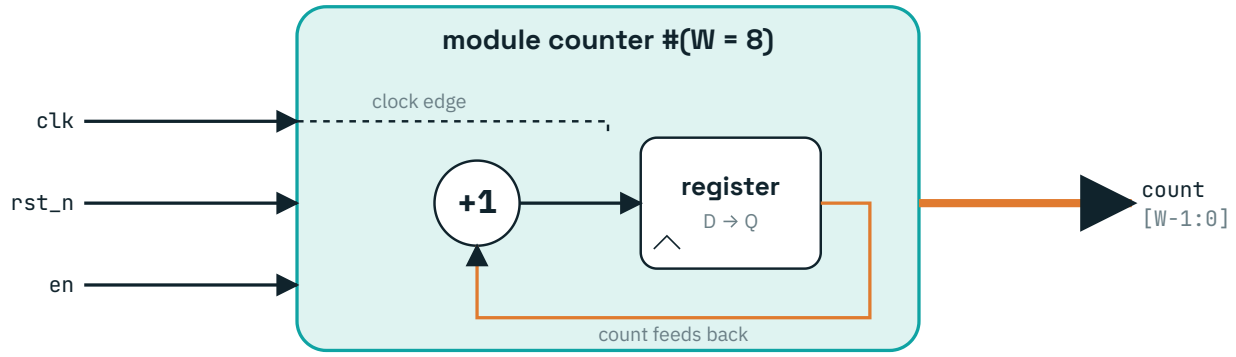
Worked example

Literal	Means	Bits
`4'b1010`	4 bits, binary	1010
`8'hFF`	8 bits, hex	1111_1111
`8'd10`	8 bits, decimal	0000_1010
`'1`	all ones, any width	1…1
`10`	unsized decimal = 32 bits	0…01010

The word for it	**Sized literals**; underscores (`8'b1011_0110`) are just for readability.
Why it matters	Unsized or wrongly sized numbers silently truncate or widen values — a common source of mismatches.

Going further: the reference

The same ideas at the next level: precise definitions, tables, rules and interview questions.



The module boundary. Ports carry signals in and out; parameters (`W`) let one description build many sizes: `counter #(.W(16)) u_cnt (...)`.

Key definitions

HDL
Hardware Description Language. Describes circuits that all operate at once, not instructions run one after another.

RTL (Register-Transfer Level)
Describing a design as registers plus the combinational logic that moves data between them on each clock edge.

Module
The unit of design in Verilog: a named block with ports. A chip is a hierarchy of module instances.

Port
A signal crossing the module boundary: `input`, `output` or `inout`.

Parameter
A compile-time constant that sizes or configures a module, overridden per instance with `#( )`.

Instance
One copy of a module placed inside another, with its own name (e.g. `u_cnt`).

Net (`wire`)
A connection driven continuously by something else; it holds no value of its own.

Variable (`reg`/`logic`)
Holds the last value assigned to it in a procedural block until assigned again.

A complete module

```
module counter #(parameter int W = 8) (
  input  wire         clk,
  input  wire         rst_n,   // active-low asynchronous reset
  input  wire         en,
  output reg  [W-1:0] count
);
  always @(posedge clk or negedge rst_n)
    if (!rst_n)  count <= '0;
    else if (en) count <= count + 1'b1;
endmodule
```

Nets versus variables

Kind	Declared as	Gets its value from	Use for
Net	`wire`	continuous drivers: `assign`, module outputs	connections between blocks
Variable	`reg` / `logic`	procedural assignment in `always` / `initial`	anything assigned in a block

reg does not mean register
A `reg` assigned in `always @*` is plain combinational logic. Only a clocked block makes a flip-flop. New code uses `logic`.

Every bit has four values

0

Logic low

Driven to ground.

1

Logic high

Driven to supply.

X

Unknown

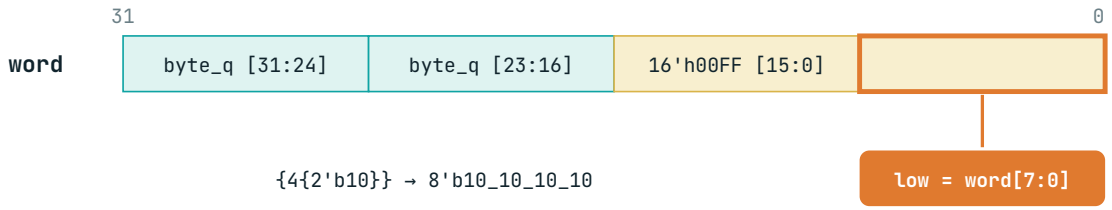
Unreset flop, two drivers fighting, undriven read.

Z

High-impedance

Nobody drives the wire (tri-state bus released).

Vectors: slice, join, repeat



Concatenation builds wide buses, part-selects take them apart. `{byte_q, byte_q, 16'h00FF}` is 32 bits, MSB on the left.

Hierarchy: modules inside modules

```
module top (input wire clk, rst_n, output wire [15:0] events);
  counter #(.W(16)) u_cnt (     // instance name: u_cnt
    .clk  (clk),
    .rst_n(rst_n),
    .en   (1'b1),
    .count(events)
  );
endmodule
```

The hierarchical path `top.u_cnt.count` names the counter's output anywhere in a simulation — testbenches and waveform viewers use these paths constantly.

Number literals you will read every day

Literal	Meaning	Value
`8'hA5`	8 bits, hexadecimal	1010_0101
`4'b10_01`	4 bits, binary; underscores are ignored	9
`16'd300`	16 bits, decimal	300
`8'sd-3` / `-8'sd3`	signed 8-bit	1111_1101
`'0`, `'1`, `'x`	fill every bit (SystemVerilog)	all zeros / ones / X
`4'bx01z`	bits can be X or Z	X 0 1 Z

Common mistakes

Connecting ports by position (breaks silently when ports are reordered) · forgetting a width so `8'hFF + 1` wraps to 0 · leaving a `wire` undriven so it reads Z · assigning a `wire` inside an `always` block (compile error).

INTERVIEW CORNER

Q. What is the difference between `wire` and `reg`? **A.** A wire is driven continuously by an assign or an output; a reg holds the value last assigned in a procedural block. Neither word decides whether hardware has a flip-flop — the clocking of the block does.

Q. When do you see Z on a wire? **A.** When nothing drives it, or every tri-state driver is released.

Key takeaways

1. A module is hardware with a boundary; instances build hierarchy.
2. Nets connect, variables hold; `reg` does not imply a register.
3. X means unknown — treat any X on a DUT output as a bug until explained.
4. Always size literals and connect ports by name.

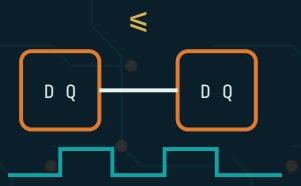
MODULE 2 OF 4

Blocking vs non-blocking

The single most important rule in RTL. See what each assignment builds, then step through the simulator's event queue.

- Mapping code to gates, muxes and flip-flops
- `=` vs `<=` with a live shift register
- The event queue, region by region
- Five rules that prevent races

⏱ 4 hours | Area 02 · Verilog



Start from zero

New to this? Work through these steps first, in order. Already comfortable? Skim them and jump to the reference below.

STEP 1

1 Three ways to write three kinds of hardware

≈ In plain words

Almost all RTL is written in three forms, and each turns into a different kind of hardware.

✳ Think of it like

Three kitchen tools: a pipe (always flowing), a mixing bowl (recalculated whenever an ingredient changes), and a fridge (stores things at set times).

✎ **Worked example**

You write	It becomes	Assign with
`assign y = …;`	gates (combinational)	—
`always @(*) …`	gates (combinational)	`=`
`always @(posedge clk) …`	flip-flops (sequential)	`<=`

Aa **The word for it** — **Continuous assignment** and **procedural blocks**.

! **Why it matters** — When you look at a line of RTL, the first question is “is this a gate or a flip-flop?” — the form tells you.

2 STEP 2 — Everything runs at the same time

≈ **In plain words** — All `assign` statements and `always` blocks run in parallel. There is no “line 1 runs before line 50”.

☀ **Think of it like** — An orchestra: every instrument plays at once, following the same conductor (the clock).

✎ **Worked example** — Two `always @(posedge clk)` blocks in different files both wake up on the same edge. Neither goes first in any meaningful sense.

Aa **The word for it** — **Concurrency**.

! **Why it matters** — This is why the order of assignments needs special rules — the next lessons.

3 STEP 3
Blocking “=”: right now, in order

In plain words

With `=` the variable changes immediately, and the next line already sees the new value.

Think of it like

Passing a note along a row: the second person reads what the first just wrote.

Worked example

1. `a = 1; b = a;`
2. After line 1, a = 1.
3. Line 2 reads the **new** a → b = 1.

The word for it

Blocking assignment.

Why it matters

Correct for combinational logic, where you calculate step by step inside one block.

4 STEP 4
Non-blocking “<=”: everyone at the same instant

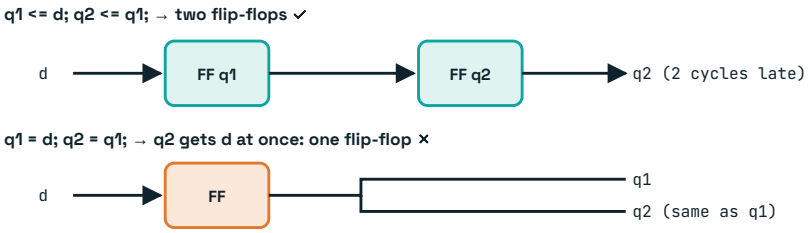
In plain words

With `<=` all right-hand sides are read first, using the **old** values; then all the left-hand sides update together.

Think of it like

A class photo: everyone freezes at the same click; nobody sees anyone else move first.

Worked example



Same two lines, different assignment, different hardware.

1. Swap with `<=` : `a <= b; b <= a;` → both read old values → a and b really swap ✓
2. Swap with `=` : `a = b; b = a;` → b reads the new a → both end up equal ×

The word for it

Non-blocking assignment (NBA).

Why it matters

Flip-flops all sample at the same edge. `<=` models exactly that, so the simulation matches the silicon.

PAUSE & TRY

At a rising edge, d = 1, q1 = 0, q2 = 0. After `q1 <= d; q2 <= q1;` what are q1 and q2? And with `=`?

Answer

With `<=`: q1 = 1, q2 = 0 (q2 got the old q1). With `=`: q1 = 1, q2 = 1 — the shift register collapsed.

STEP 5

5 Five rules that keep you safe

In plain words

Follow these and races between blocks cannot happen.

Think of it like

Traffic rules: they look strict, but everyone arrives safely.

✎ **Worked example**

1. Flip-flops (`always @(posedge clk)`): use `<=` .
2. Combinational (`always @(*)`): use `=` .
3. Don't mix `=` and `<=` in one block.
4. Assign each variable from only one `always` block.
5. Use `@(*)` so the block wakes on every input.

Aa **The word for it**

Race condition: a result that depends on which block the simulator happens to run first.

! **Why it matters**

Races make tests pass on one simulator and fail on another. Reviewers check these rules first.

6 STEP 6 Inside one time step

≈ **In plain words**

At each moment the simulator first runs everything that's ready (blocking assignments, gate updates), and only **then** applies all the `<=` updates.

☀ **Think of it like**

A teacher collects every answer sheet first, and only then marks them — so no one can copy a marked answer.

✎ **Worked example**

1. Clock rises.
2. All flip-flop blocks wake and read their inputs (old values).
3. Their `<=` updates are queued.
4. Queued updates are applied together; gates re-evaluate.

Aa **The word for it**

The **event queue**: active region, then the NBA region.

! **Why it matters**

This explains every “why did my signal change one cycle late?” question you will meet.

7 STEP 7 A testbench race, and how to avoid it

In plain words — If the testbench changes an input with `=` exactly at the clock edge, the design may see the old or the new value — it depends on the order the simulator picks.

Think of it like — Two people grabbing the same door handle at the same instant.

Worked example

```
// risky: same edge, blocking
@(posedge clk) din = 1;
// safe: drive with <= (or on the falling edge)
@(posedge clk) din <= 1;
```

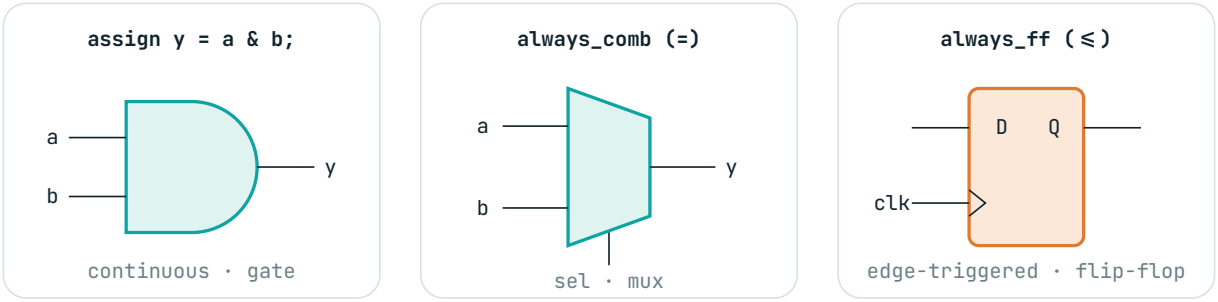
The word for it — A **testbench race**.

Why it matters — Drive inputs away from the sampling edge. It's the first habit of a reliable testbench.

Going further: the reference

The same ideas at the next level: precise definitions, tables, rules and interview questions.

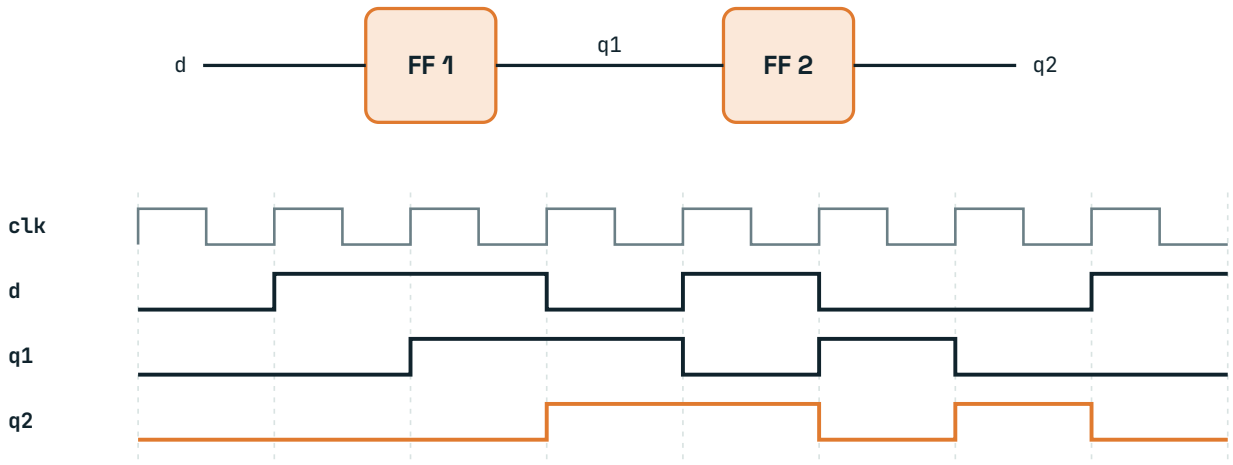
Three kinds of hardware, three ways to write them



Pick the block by the hardware you want. Combinational logic uses blocking =; flip-flops use non-blocking <=.

Try it: a two-stage shift register

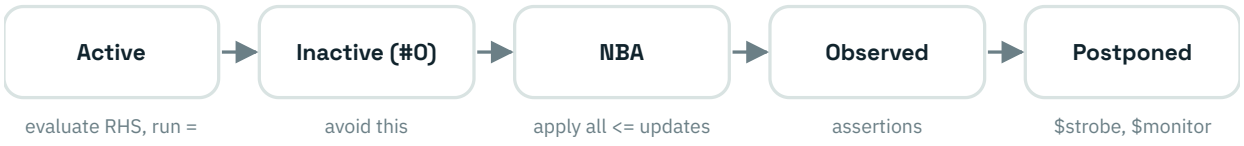
```
always @(posedge clk) begin
  q1 <= d;
  q2 <= q1;   // reads the OLD q1
end
```



Two stages. q2 is q1 delayed by one more clock: every flop samples old values, then all update together, exactly like real hardware.

Inside one time step: the event queue

Within a single simulation time the simulator runs **regions** in a fixed order. Step through what happens to `count <= count + 1` at a clock edge.




```
count = 4 before the edge. Press Step.
```

★ **Five rules that prevent races**

1 Sequential: `<=` only. 2 Combinational: `=` only, default every output first. 3 One variable, one `always` block. 4 Never mix `=` and `<=` on a variable. 5 Testbenches drive inputs away from the active edge.

Key definitions

Blocking assignment =
Updates the left side immediately; the next statement in the block sees the new value.

Non-blocking assignment <=
Evaluates the right side now and schedules the update for the NBA region of the same time step.

Time step
All the activity at one simulation time. Many events and regions can run inside it with zero time passing.

Delta cycle
One pass through the active regions inside a time step; updates can trigger more deltas before time advances.

Race condition
A result that depends on the order the simulator happens to run two processes in — legal for the simulator, unpredictable for you.

Sensitivity list
What wakes a block: `@(posedge clk)`, `@*` (every input it reads).

Worked example: a testbench race

```
// DUT samples d at posedge clk
always @(posedge clk) q <= d;

// Testbench A — RACE: changes d with = exactly at the edge
initial begin @(posedge clk); d = 1; end

// Testbench B — safe: drive on the falling edge
initial begin @(negedge clk); d = 1; end
```

In A, whether `q` captures 0 or 1 at that edge depends on which process the simulator runs first. Two simulators — or the same one after a code change — can disagree. B moves the change half a cycle away, so there is nothing to race.

Common mistakes

Using `=` in clocked blocks · incomplete sensitivity lists like `always @(a)` when the block also reads `b` (simulation and synthesis then disagree; use `@*`) · driving the same signal from two always blocks · adding `#1` delays to "fix" a race instead of removing it.

INTERVIEW CORNER

Q. Swap two registers without a temp variable. **A.** `always @(posedge clk) begin a <= b; b <= a; end` — both right sides are sampled before either update.

Q. Why is `#0` discouraged? **A.** It only pushes an event to the Inactive region; it reorders a race instead of removing it.

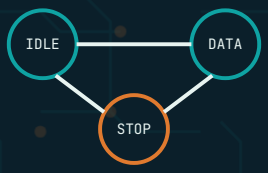
Key takeaways

1. Flip-flops: `<=`. Combinational logic: `=` with `@*`.
2. Inside one time step, regions run in a fixed order; NBA updates come after all Active evaluation.
3. Races come from ordering; fix them with coding style, never with delays.

MODULE 3 OF 4

Synthesizable state machines

The UART transmitter you test in the lab is a four-state machine. Read it as a diagram first, then as code.



- Moore vs Mealy, encoded with `enum`
- The two-process FSM template
- Synthesizable vs simulation-only constructs
- Reset strategy that verification can trust

⏱ 3.5 hours · Area 02 · Verilog

Start from zero

New to this? Work through these steps first, in order. Already comfortable? Skim them and jump to the reference below.

STEP 1

A state machine, coded

In plain words

From Area 01: an FSM is a **state register** plus **next-state logic**. In Verilog that becomes one clocked block and one combinational block.

Think of it like

A board game: the counter on the board (state) and the rule card that says where to move next (logic).

Worked example

```
always @(posedge clk) if (rst) state <= IDLE; else state <
always @(*) begin next = state; case (state) … endcase end
```

The word for it

The **two-process FSM** template.

! **Why it matters** Every protocol controller you verify is built this way, so you'll recognise the structure instantly.

2 STEP 2 Give states names

≈ **In plain words** States are just numbers in hardware, but you should give them names.

✳ **Think of it like** Rooms called “Kitchen” and “Bedroom” instead of “Room 2” and “Room 3”.

✎ **Worked example**

```
localparam IDLE = 2'd0, START = 2'd1, DATA = 2'd2, STOP =
// SystemVerilog: typedef enum logic [1:0] {IDLE, START, D
```

Aa **The word for it** **State encoding**: binary, one-hot or Gray (details below).

! **Why it matters** Waveform viewers show `enum` names, so you read “DATA” instead of decoding `2` in your head.

3 STEP 3 Moore or Mealy?

≈ **In plain words** A **Moore** output depends only on the state. A **Mealy** output depends on the state **and** the current input.

✳ **Think of it like** Moore: a sign that says “READY” while the machine is in the ready state. Mealy: the snack drops the very moment you press the button.

✎ **Worked example**

	Moore	Mealy
Output from	state	state + input
Reacts	next cycle	same cycle
Glitches on output	no	can follow input glitches

Aa **The word for it** — **Moore machine**, **Mealy machine**.

! **Why it matters** — It decides **when** your checker should expect the output: one cycle later, or immediately.

4 STEP 4 — The accidental latch

≈ **In plain words** — In a combinational block, every output must get a value on every path. If one path forgets, the tool builds a **latch** to remember the old value.

☀ **Think of it like** — A door with no spring: if nobody pushes it, it stays wherever it was last left.

✎ **Worked example**

```
always @(*) if (en) y = a;          // en = 0: y not assig
always @(*) begin y = 0; if (en) y = a; end   // default f
```

Aa **The word for it** — An **inferred latch**; the fix is a **default assignment** (`next = state;`) at the top.

! **Why it matters** — Synthesis and lint tools warn about latches; you'll see these warnings in reports and must know what they mean.

5 STEP 5 — What becomes hardware, and what doesn't

≈ **In plain words** Some Verilog only makes sense in simulation and is ignored (or rejected) by synthesis.

☀ **Think of it like** Stage directions in a script: the actors read them, but the audience never hears them.

✎ **Worked example**

Code	In silicon?
`assign`, `always @(*)`, `always @(posedge clk)`	Yes
`#5` delays	No — ignored
`$display`, `$finish`	No
`initial` blocks	No (except some FPGA init)

Aa **The word for it** **Synthesizable** versus **simulation-only** constructs.

! **Why it matters** Testbenches use everything; RTL may use only the synthesizable part. A `#5` in RTL is a red flag in review.

PAUSE & TRY **Which of these end up as hardware: `assign` , `#10` , `$display` , `always @(posedge clk)` ?** Answer

Only `assign` and `always @(posedge clk)`.

6 STEP 6

A reset you can trust

≈ **In plain words** Every state register needs a reset that puts the FSM in a known state (usually IDLE).

Think of it like

Pressing “home” on a game controller: wherever you were, you're back at the menu.

Worked example

Style	Code	Acts
Synchronous	`always @(posedge clk) if (rst) …`	on the next clock edge
Asynchronous	`always @(posedge clk or posedge rst) if (rst) …`	immediately

The word for it

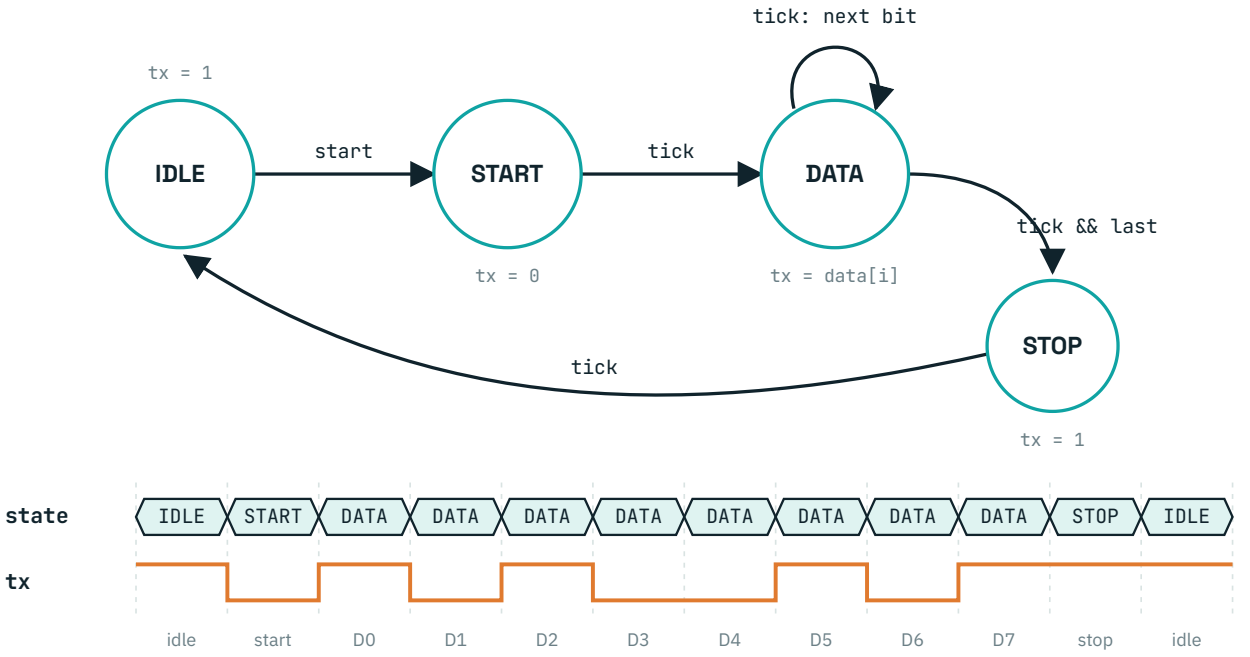
Synchronous and **asynchronous reset**.

Why it matters

The first check of any FSM test: after reset, the state is IDLE and the outputs are idle values.

Going further: the reference

The same ideas at the next level: precise definitions, tables, rules and interview questions.



One frame, four states. The line idles high, drops for the start bit, sends eight data bits **LSB first**, then a high stop bit. Press Send to watch the machine and the wire move together.

```
typedef enum logic [1:0] {IDLE, START, DATA, STOP} state_t;
state_t state, next;

always_ff @(posedge clk or negedge rst_n)       // 1) state register
  if (!rst_n) state <= IDLE; else state <= next;

always_comb begin                                // 2) next-state logic
  next = state;                                   // default: stay — no latches
  case (state)
    IDLE:  if (start)        next = START;
    START: if (tick)         next = DATA;
    DATA:  if (tick && last) next = STOP;
    STOP:  if (tick)         next = IDLE;
  endcase
end
```

Synthesizable	Simulation only
`assign`, `always_comb`, `always_ff`	`initial`, `#delays`, `wait`
`if`, `case`, `for` with constant bounds	`$display`, `$random`, file I/O
arithmetic, shifts, compares, parameters	`force` / `release`

Key definitions

Finite state machine (FSM)
Logic with a state register and rules that pick the next state from the current state and inputs.

Moore machine
Outputs depend only on the current state. Glitch-free, one cycle later to react.

Mealy machine
Outputs depend on state and inputs. Reacts in the same cycle; outputs can glitch with inputs.

Latch
Level-sensitive storage, usually created by accident when a combinational block does not assign an output on every path.

Synthesis
Turning RTL into a gate-level netlist. Only a subset of Verilog (the synthesizable subset) can be turned into gates.

Reset
Forces registers to a known value. Asynchronous resets act immediately; synchronous ones on the next clock edge.

State encodings

Encoding	4 states as	Flip-flops	Good for
Binary	00, 01, 10, 11	2	Smallest area
One-hot	0001, 0010, 0100, 1000	4	Fast decode; common on FPGAs
Gray	00, 01, 11, 10	2	One bit changes per step; counters crossing clock domains

★ **Verification habit: check after reset**

Right after reset is released, assert that every output is known: `if ($isunknown(tx)) $error("tx is X after reset");`. Unreset state is one of the most common silicon bugs.

× **Common mistakes**

Missing `default` / default assignment → latch · next-state logic in the clocked block (works, but hard to read and to cover) · forgetting to reset the state register · comparing state to raw numbers instead of enum names.

INTERVIEW CORNER

Q. How would you detect an unintended latch? **A.** Synthesis and lint (Verilator `-Wall`) warn about inferred latches; in review, look for combinational blocks without default assignments.

Q. Moore or Mealy for a UART transmitter? **A.** Moore: the line value must be stable for a whole bit period, so it should depend only on state.

Key takeaways

1. Two processes: a clocked state register and a combinational next-state block.
2. Default the next state at the top of `always_comb` .
3. Keep delays, prints and `initial` out of RTL.

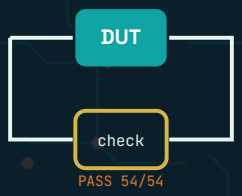
MODULE 4 OF 4

Your first testbench

Every testbench — from this lab to a full UVM environment — has the same four jobs. Build them as separate pieces from day one.

- Clock, reset and stimulus
- Driver, monitor, reference model, checker
- Sampling a serial line correctly
- Self-checking PASS / FAIL reporting

⏱ 4.5 hours | Area 02 · Verilog



Start from zero

New to this? Work through these steps first, in order. Already comfortable? Skim them and jump to the reference below.

STEP 1

What a testbench is

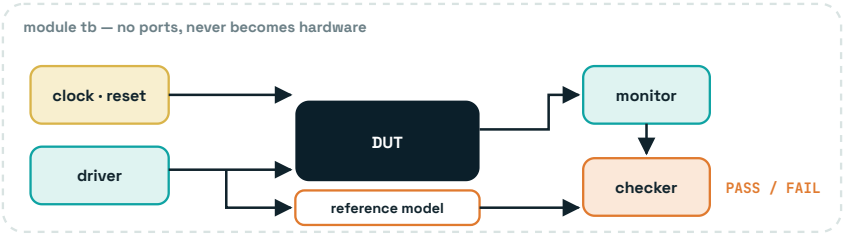
≈ **In plain words**

A testbench is Verilog code that surrounds your design, feeds it inputs and checks its outputs. It has no ports and never becomes hardware.

☀ **Think of it like**

A car on a test bench in a garage: rollers spin the wheels, sensors read the engine — the car doesn't know it isn't on a road.

✎ **Worked example**



The parts of every self-checking testbench.

Aa **The word for it** — **Testbench**; the design inside it is the **DUT**.

! **Why it matters** — This picture is the skeleton of everything you'll build — up to full UVM environments in later areas.

2 STEP 2 — Making a clock and a reset

≈ **In plain words** — The testbench creates the clock by flipping a signal forever, and holds reset for a few cycles at the start.

☀ **Think of it like** — Starting a metronome, then counting “one, two” before the band starts playing.

✎ **Worked example**

```
reg clk = 0, rst = 1;
always #5 clk = ~clk;              // period 10 ns → 100 MHz
initial begin
  repeat (3) @(posedge clk);    // 3 cycles in reset
  rst <= 0;
end
```

Aa **The word for it** — **Clock generation** and the **reset sequence**.

! **Why it matters** — Wrong clock periods and too-short resets are behind many “the design does nothing” first runs.

PAUSE & TRY **With `always #5 clk = ~clk;` in a 1 ns timescale, what is the clock frequency?** Answer

High 5 ns + low 5 ns = 10 ns period → **100 MHz**.

3 STEP 3 Driving inputs

≈ In plain words A driver sets the DUT's inputs, cycle by cycle. Wrap repeated actions in a `task`.

☀ Think of it like A robot arm pressing buttons in a fixed rhythm.

✎ Worked example

```
task send(input [7:0] b);
  @(posedge clk); data <= b; valid <= 1;
  @(posedge clk); valid <= 0;
endtask
initial begin send(8'h41); send(8'h03); end
```

Aa The word for it A **driver** and a **task**.

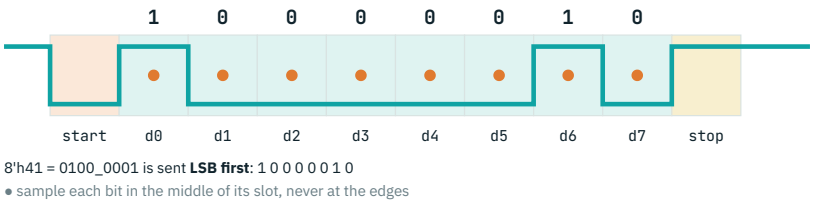
! Why it matters Tasks turn stimulus into readable sentences — “send A, then send 3” — which you'll extend into sequences later.

4 STEP 4 Sampling a serial line

≈ In plain words A UART sends one byte as a **frame**: a start bit (0), eight data bits **least significant first**, then a stop bit (1). A monitor rebuilds the byte by sampling each bit in the **middle** of its time slot.

☀ Think of it like Reading a ticker tape: you read each letter in the centre of its box, not on the line between boxes.

Worked example



8'h41 = 0100_0001 is sent **LSB first**: 1 0 0 0 0 0 1 0
• sample each bit in the middle of its slot, never at the edges

How 8'h41 (the letter “A”) appears on the wire.

The word for it

Frame, **LSB-first**, **mid-bit sampling**.

Why it matters

Lab 02's planted bug sends bits in the wrong order. A correct monitor is what catches it.

PAUSE & TRY **Write the line bits (start to stop) for `8'h03`, sent LSB first.** Answer

`8'h03` = `0000_0011` → data bits LSB first are 1 1 0 0 0 0 0 0. Frame: **0 · 1 1 0 0 0 0 0 0 0 · 1**.

5 STEP 5 Checking against a reference model

In plain words

A reference model computes what the output **should** be in plain code. The checker compares the DUT's result with it and counts mismatches.

Think of it like

Marking a test with the answer key.

Worked example

```
expected = model(byte_sent);
if (got !== expected) begin
  errors = errors + 1;
  $display("MISMATCH sent %h got %h", expected, got);
end
```

Aa The word for it

Reference model, **checker**. Use !==: it treats X and Z as differences, while != would hide them.

! Why it matters

A testbench that only drives inputs proves nothing. Self-checking is what makes it a test.

6 STEP 6 PASS or FAIL, honestly

≈ In plain words

At the end, print one clear line: how many checks ran and whether all passed.

✳ Think of it like

A scoreboard at the end of a match — no one has to watch the whole game to know the result.

✎ Worked example

```
if (checks == 0)        $display("FAIL: no checks ran");
else if (errors == 0) $display("PASS %0d/%0d", checks, che
else                    $display("FAIL %0d errors", errors
```

Aa The word for it

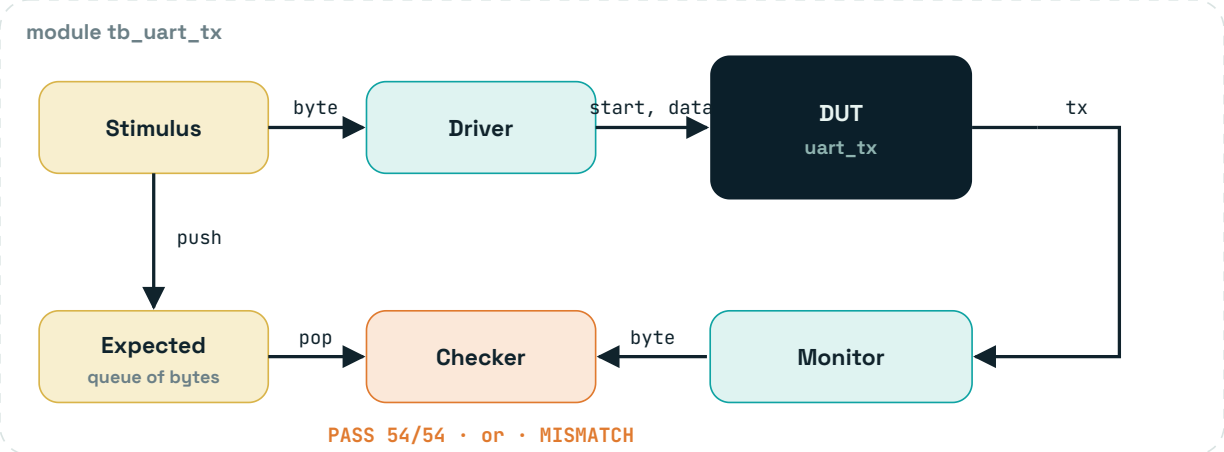
A **self-checking testbench**.

! Why it matters

“PASS with zero checks” is a silent failure. Always prove that the checks actually ran.

Going further: the reference

The same ideas at the next level: precise definitions, tables, rules and interview questions.



Four parts, one loop. The driver turns transactions into pin wiggles, the monitor turns pins back into transactions, and the checker compares them with what was sent. UVM formalises exactly these parts in Areas 08–12.

Part	Job in this lab	Becomes in UVM
Driver	`send(b)` task pulses `start` with `data`	`uvm_driver`
Monitor	finds the start bit, samples mid-bit, rebuilds the byte	`uvm_monitor`
Reference model	queue of bytes in send order	predictor
Checker	compares, counts, prints PASS / FAIL	`uvm_scoreboard`

Sample in the middle of the bit

After the falling edge of the start bit, wait `CLKS_PER_BIT/2` clocks, confirm the line is still 0, then sample every `CLKS_PER_BIT` clocks. The middle has the most margin against timing error.

Key definitions

DUT
Design Under Test: the module being verified, instantiated inside the testbench.

Testbench
A top-level module with no ports that creates clocks, drives inputs, observes outputs and reports results.

Transaction
A unit of work at a higher level than pins, such as "send byte 0xA5".

Self-checking
The testbench decides PASS or FAIL itself, so nobody has to read waveforms to know the result.

Reference model
An independent prediction of what the DUT should do — here simply the list of bytes sent.

Regression
Running many tests, usually every night or on every change, to catch anything that broke.

The complete monitor (the pattern you will write in the lab)

```
initial begin
  @(posedge rst_n);
  forever begin
    @(negedge tx);                          // start bit begins
    repeat (CPB/2) @(posedge clk);            // move to its middle
    if (tx !== 1'b0) $error("false start");
    for (i = 0; i < 8; i = i + 1) begin
      repeat (CPB) @(posedge clk);
      rx[i] = tx;                               // LSB first
    end
    repeat (CPB) @(posedge clk);
    if (tx !== 1'b1) $error("bad stop bit");
    if (rx !== expected[n_checked]) n_errors = n_errors + 1;
    n_checked = n_checked + 1;
  end
end
```

System tasks every testbench uses

Task	Does
`$display` / `$write`	Print now (with / without newline)
`$strobe` / `$monitor`	Print at the end of the time step / whenever an argument changes
`$time` , `%0t`	Current simulation time and how to print it
`$random` , `$urandom`	Random numbers for stimulus
`$dumpfile` , `$dumpvars`	Write a VCD waveform file
`$error` , `$fatal`	Report a failure (and stop, for $fatal)
`$finish`	End the simulation

Common mistakes

Ending the test before the last byte is checked · printing PASS when zero items were checked · comparing with `!=` so X slips through · sampling exactly on an edge.

INTERVIEW CORNER

Q. What makes a testbench self-checking? **A.** An independent expected result, an automatic comparison, and a summary that fails loudly — including when nothing was checked.

Q. Why separate driver and monitor? **A.** So the monitor checks what really appeared on the pins, not what the driver intended; it also lets you reuse the monitor when another block drives the bus.

Key takeaways

1. Driver, monitor, reference model, checker — even in a 50-line testbench.
2. Compare with `!==`; count checks; fail when nothing was checked.
3. Sample serial data in the middle of the bit.

PRACTICE

Check yourself

Answer in your head, then open each card. The area test has 20 more.



Why doesn't `reg` imply a flip-flop?

`reg` only means "assigned in a procedural block". The block's sensitivity decides the hardware: `always @*` gives logic, `always @(posedge clk)` gives flops.

After one edge in the blocking shift register, what does q2 hold?

The new `d`. Blocking updates `q1` immediately, so `q2 = q1` reads the new value and the two stages collapse into one.

Why does `$strobe` print the new value but `$display` the old one?

`$display` runs in the Active region, before the NBA region applies <= updates. `$strobe` runs in Postponed, after everything settled.

Name two reasons an output can be X after reset.

A register feeding it has no reset and was never written; or two drivers fight over the same net.

What hardware does `assign y = sel ? a : b;` build?

A 2:1 multiplexer: `sel` chooses which input reaches `y`.

A combinational `always @*` block assigns `y` only inside `if (en)`. What happens?

When `en` is 0, `y` must keep its value, so synthesis infers a latch. Add `y = '0;` (or a default) before the `if`.

How many clocks does one UART frame take at CLKS_PER_BIT = 8?

10 bits (start + 8 data + stop) × 8 = 80 clocks.

Your testbench prints PASS but checked 0 bytes. Why is that a bug?

A test that checks nothing passes for any design. Always require `n_checked == n_sent` and `n_checked > 0`.

REFERENCE

Interview questions

Asked in DV and RTL interviews at every level. Try an answer out loud before opening.



Explain blocking vs non-blocking with an example.

Show the two-stage shift register: with <= you get two flops, with = the second stage collapses. Then state the rule: <= in sequential, = in combinational blocks.

What is the difference between `==` and `===` ?

== returns X if either operand has X or Z bits; === compares X and Z literally and always returns 0 or 1. Testbenches use ===/!==.

How does synthesis treat an `initial` block?

ASIC synthesis ignores it. Some FPGA tools use it to set power-up register values. Never rely on it for ASIC reset.

What is a full case / parallel case problem?

If a `case` doesn't cover every value and there's no default, combinational logic infers a latch. Synthesis pragmas (`full_case`, `parallel_case`) can make synthesis and simulation disagree — prefer a default and SystemVerilog `unique`/`priority`.

Write a divide-by-2 clock.

`always @(posedge clk or negedge rst_n) if (!rst_n) clk2 <= 0; else clk2 <= ~clk2;` — and mention that real designs prefer clock enables over generated clocks.

Asynchronous vs synchronous reset?

Async acts immediately (in the sensitivity list), needs a synchronised release; sync acts on the clock edge and needs the clock running. Both are common; follow the project's convention.

What does `$monitor` do differently from `$display` ?

`$monitor` prints whenever any of its arguments changes, at the end of the time step; only one is active at a time.

How do you make a testbench fail if the DUT hangs?

Add a watchdog: `initial begin #1ms; $fatal(1, "timeout"); end`, or a cycle counter that errors if `busy` stays high too long.

REFERENCE

Glossary

Every term used on this page, A to Z.



Active region
Where processes run and blocking assignments update within a time step.

Asynchronous reset
Reset in the sensitivity list; takes effect without a clock edge.

Combinational logic
Outputs depend only on current inputs; no memory.

Concatenation
`{a, b}` joins vectors into a wider one.

Delta cycle
A zero-time pass through the scheduler inside one time step.

DUT
Design under test.

Event queue
The simulator's ordered list of things to do at each time.

Flip-flop
Edge-triggered storage element; built from `always @(posedge clk)`.

FSM
Finite state machine: state register + next-state logic.

Instance
A named copy of a module inside another module.

Latch
Level-sensitive storage, often inferred by mistake.

Lint
Static checks for suspicious code (Verilator `--lint-only -Wall`).

Monitor
Testbench part that turns pin activity into transactions.

NBA region
Where scheduled non-blocking updates are applied.

Net
A continuously driven connection (`wire`).

Netlist
The gate-level design produced by synthesis.

Parameter
Compile-time constant that configures a module.

Part-select
`v[7:0]`: a contiguous slice of a vector.

Race condition
Behaviour that depends on process execution order.

RTL
Register-transfer level: registers plus the logic between them.

Self-checking testbench
Decides PASS/FAIL automatically.

Synthesis
Converting RTL into gates.

UART 8N1
Serial format: 8 data bits, no parity, 1 stop bit, LSB first.

VCD
Value Change Dump: the standard waveform file format.

X / Z
Unknown / high-impedance values in four-state logic.

REFERENCE

FAQ & troubleshooting

The errors learners hit most in this area, and the fix for each.



You see	Why	Fix
`iverilog: command not found`	Not installed, or not on PATH	Re-run the install step; on Windows reopen the terminal after the installer
`Unknown module type: uart_tx`	The design file wasn't passed to the compiler	List every file: `iverilog -o sim.vvp uart_tx.v tb_uart_tx.v`
`syntax error` on `logic` or `always_ff`	SystemVerilog keywords without SV mode	Add `-g2012` (Icarus) or name the file `.sv`
Simulation never ends	No `$finish`, or waiting on a signal that never changes	Add a watchdog timeout; check the `wait` condition
Output is X forever	Reset never asserted, or a register isn't reset	Hold `rst_n` low for a few clocks at the start
Waveform file is empty	No `$dumpvars`, or `$finish` before anything happened	Add the dump block at time 0
Results differ between Icarus and Verilator	A race or X-dependent code	Apply the five race rules; check for X with `$isunknown`

REFERENCE

Further reading

Go deeper when you are ready. Standards are the final word on simulator behaviour.



Resource	Why read it
IEEE 1364-2005 (Verilog) and **IEEE 1800** (SystemVerilog) standards	The scheduling regions and every language rule, defined exactly
Clifford E. Cummings, *Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!* (SNUG 2000)	The classic paper behind the race-free coding rules
Samir Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*	Thorough language reference with exercises
Harris & Harris, *Digital Design and Computer Architecture*	Digital logic and HDL side by side
Icarus Verilog and Verilator user guides	Command-line options, warnings and lint flags

HANDS-ON LAB · ABOUT 3 HOURS

Lab 02 · UART transmitter and its first testbench

You get `uart_tx.v` (8N1, one planted bug), `tb_uart_tx.v` with a driver and an empty monitor, and a Makefile for Icarus Verilog or Verilator.

1. Complete the monitor: detect the start bit, sample each bit mid-period, check the stop bit, rebuild the byte.
2. Compare every received byte with the expected queue.
3. Send 0x00, 0xFF, 0xA5, 0x5A and 50 random bytes — the mismatches reveal the bug's pattern.

4. Fix the RTL and re-run until you see `PASS: 54/54 bytes`.