3.7 HDL and verification basics

Name: _____ Class: _____ Date: _____ **Total: 21 marks**

KEY WORDS hardware description language • synthesizable • testbench • simulation • synthesis

Objective

Write synthesizable HDL for combinational and sequential blocks, write testbenches that check them exhaustively, and keep the simulation/synthesis boundary crisp.

1 Key knowledge and worked examples

Synthesizable describes hardware

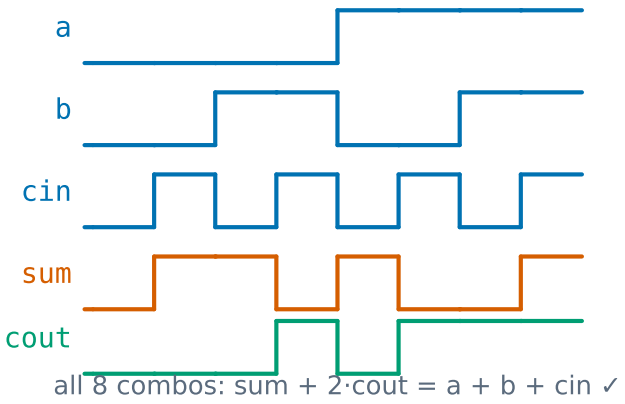
A **hardware description language** (Verilog/VHDL) describes structure and behaviour. **Synthesizable** code becomes gates: `assign` expressions, `always @(*)` combinational, `always @(posedge clk)` sequential, `case`/`if` structure. Non-synthesizable constructs (`#delay`, `initial` in hardware, waits) belong only to the **testbench** – the simulation-only harness that drives inputs and checks outputs.

```
module full_adder(input a, b, cin, output sum, cout);
  assign sum  = a ^ b ^ cin;
  assign cout = (a & b) | (cin & (a ^ b));
endmodule
```

One module, one exhaustive testbench

```
module full_adder(
  input a, b, cin,
  output sum, cout);
  assign sum =
      a ^ b ^ cin;
  assign cout =
      (a & b) |
      (cin & (a^b));
endmodule
```

synthesizable module



Worked example: the exhaustive testbench

```
module tb;
  logic a, b, cin, sum, cout;
  full_adder dut(.a(a), .b(b), .cin(cin), .sum(sum), .cout(cout));
  initial begin
    for (int i = 0; i < 8; i++) begin
      {a, b, cin} = i; #10;
      assert (2*cout + sum == a + b + cin)
        else $error("fail at %b", {a,b,cin});
    end
    $display("all 8 combinations pass");
  end
endmodule
```

Eight loops, one arithmetic invariant – the same exhaustive check the paper-and-Karnaugh route used in unit 3.2, now automatic. **Simulation** runs in time steps; **synthesis** maps the design to gates.

The boundary rules

- One driver per signal in synthesizable code (`assign` XOR `always`, not both).
- Sequential always blocks infer flip-flops – every path that leaves a value unchanged implies memory (that is how registers are meant to be inferred).
- The testbench may freely use delays, loops and `initial` – it never becomes hardware, so those constructs are safe THERE and only there.

2 Practice

2.1 Classify each as synthesizable or testbench-only: `#10;`, `assign y = a & b;`, `$display("hi");`, `always @(posedge clk) q <= d;`. [4]

2.2 Write one line of Verilog assigning `y` as the XOR of three inputs a, b, c. [2]

2.3 What does an `always @(posedge clk)` block whose body assigns `q <= d` infer in hardware? [2]

3 Exam-style questions

3.1 Write a synthesizable Verilog module `mux4` (4-to-1 multiplexer: inputs `d[3:0]`, selects `s[1:0]`, output `y`) using a `case` statement.

Then write the core of an exhaustive testbench for it (loop over the 16 data × 4 select combinations is 64 cases; describe the loop and the expected-value check using the indexing `d[s]`). State why `d[s]` in the testbench is legitimate even though it never appears in the hardware. [7]

3.2 A module has:

```
always @(posedge clk) begin
  if (en) q <= d;
end
```

Explain what hardware this infers and why (the unchanged path implies memory). Then extend it to a 4-bit **left** shift register with serial input `sin` and enable `en`, and describe a three-cycle test that distinguishes it from a register that merely loads. [6]