BLINKNBUILD ELECTRONICS · EMBEDDED SYSTEMS · MAKER CONTENT

Verilog & SystemVerilog — Master Cheat Sheet

Zero → Advanced RTL · Simulation semantics · Synthesis · CDC / STA · SystemVerilog & UVM · 110+ interview Q&A, traps and code puzzles

DIFFICULTY E easy M medium H hard // IEEE 1364-2005 · IEEE 1800-2023 // CODE: keyword · type · literal · $system · comment

// §1 LANGUAGE CORE — SYNTAX FROM SCRATCH

1.1 Module, Ports & Instantiation

```
// ---- ANSI style (Verilog-2001+) : PREFERRED ----
module adder #(parameter W = 8, parameter LAT = 1)
( input  wire          clk, rst_n,
  input  wire [W-1:0] a, b,
  input  wire          cin,
  output reg  [W-1:0] sum,
  output wire          cout );
  ...
endmodule

// ---- Non-ANSI (Verilog-1995 legacy, still seen in IP) ----
module adder (clk, a, b, sum);
  parameter W = 8;
  input          clk;
  input  [W-1:0] a, b;
  output [W-1:0] sum;
  reg    [W-1:0] sum;        // separate type declaration
endmodule

// ---- Instantiation ----
adder #(.W(16), .LAT(2)) u_add (    // named params  <-- ALWAYS
   .clk(clk), .rst_n(rst_n),        // named ports   <-- ALWAYS
   .a(op_a), .b(op_b), .cin(1'b0),
   .sum(result), .cout());          // unconnected output = legal
adder #(16) u2 (clk,rst_n,a,b,c,s,co);   // positional - AVOID
```

Port	Type inside module	Connection outside
input	net (wire) only, read-only	net or variable
output	wire (assign) or reg (always)	**net only** in Verilog
inout	wire only, tri-state driven	net only

Trap: a misspelled port name silently creates an implicit 1-bit `wire`. Put `` `default_nettype none `` at the top of every file (and `` `default_nettype wire `` at the bottom) so typos become compile errors.

1.2 Literals & Numbers

```
<size>'<signed><base><value>       // base: b B o O d D h H
8'b1010_0101   8'hA5   8'd165   8'o245    // all equal 165
4'bx   4'bz   4'b1x0z                    // 4-state literals
'0 '1 'x 'z                              // SV: fill ALL bits
16'shFFFF                                // signed hex = -1
'd10                                     // unsized -> 32 bits
{4{2'b01}}                               // replicate -> 8'b01010101
{a, b[3:0], 2'b11}                       // concatenation
```

- Unsized literal = **32 bits, unsigned** (unless `'sd`).
- `_` is a free separator (not leading).
- Value narrower than size → zero-padded; but padded with `x` / `z` if MSB is `x` / `z`.
- Value wider than size → **silently truncated** from MSB.
- `-8'd5` negates the value 5 → `8'b1111_1011`.

1.3 Data Types — `wire` vs `reg` vs `logic`

Feature	`wire` (net)	`reg` (var)	`logic` (SV)
Driven by	assign, port, primitive	always/initial	either, but only **one** source
Hardware	connection, no storage	comb net **or** flop — context decides	same as reg, better name
Multi-driver	allowed → resolution	illegal	compile error
Default value	1'bz	1'bx	1'bx

The #1 myth: "`reg` means a register." **False.** `reg` is only a variable that holds its value between assignments in simulation. `reg` inside `always @*` synthesises to pure combinational logic — only an **edge** in the sensitivity list creates a flip-flop.

Type	Width / states	Sign	Notes
reg / logic	1 bit, 4-state	unsigned	`reg signed [7:0]` to make signed
integer	32 bit, 4-state	**signed**	loop counters, generate indices
time	64 bit, 4-state	unsigned	holds $time
real / realtime	C double	signed	non-synthesisable
bit (SV)	1 bit, **2-state**	unsigned	faster sim, hides x-propagation
byte/shortint/int/longint	8/16/32/64, 2-state	**signed**	SV C-like types
string, event, chandle	—	—	verification only

1.4 Net Types & Drive-Strength Resolution

Net	Behaviour with multiple drivers
wire / tri	0+1 → x; z+v → v; equal → same
wand / triand	wired-AND of all drivers
wor / trior	wired-OR of all drivers
tri0 / tri1	pulls to 0 / 1 when every driver is z
trireg	charge storage (capacitive node)
supply0 / supply1	constant GND / VDD, strongest
uwire	SV single-driver net, checked at compile

Strength, weak→strong: `highz < small < medium < weak < large < pull < strong < supply`. Stronger wins; equal strength + opposite value → `x`.

1.5 Vectors, Part-Selects & Arrays

```
reg [7:0] a;            // little-endian, a[7]=MSB   <-- USE THIS
reg [0:7] b;            // big-endian, b[0]=MSB      (never mix)
a[3]                    // bit-select (out of range -> x)
a[5:2]                  // constant part-select
a[i +: 4]               // indexed: a[i+3 : i]  (width const, i variable)
a[i -: 4]               // indexed: a[i : i-3]
{a[3:0], a[7:4]}        // nibble swap

// packed = one contiguous vector (a bus)
logic [3:0][7:0] pk;    // 32 contiguous bits; pk[2][6] legal
// unpacked = array of objects (a memory)
reg  [7:0] mem [0:255]; // 256 words x 8 bits
reg  [7:0] mem2 [256];  // SV shorthand, identical
logic [7:0] m3 [4][8];  // 2-D unpacked

$bits(a) $size(a) $left(a) $right(a) $high(a) $low(a) $dimensions(a)
```

Packed vs unpacked: packed dimensions go **before** the name, are contiguous bits, and can be sliced or assigned as one vector. Unpacked dimensions go **after** the name, model memories, cannot be part-selected as a whole, and in plain Verilog cannot be passed through a port or copied wholesale.

1.6 Operators & Precedence (top binds tightest)

Lvl	Operators	Notes
1	() [] :: .	group, select, scope, member
2	+ - ! ~ & ~& | ~| ^ ~^ ++ --	unary: sign, NOT, **reduction**, inc/dec
3	**	power
4	* / %	% takes sign of 1st operand; / truncates toward 0
5	+ -	binary add / sub
6	<< >> <<< >>>	logical / **arithmetic** shift (>>> sign-extends only if LHS signed)
7	< <= > >=	relational, 1-bit result, x anywhere → x
8	== != === !== ==? !=?	logical / **case** / wildcard equality
9	& then ^ ~^ then |	binary bitwise — three separate levels
10	&& then ||	logical, 1-bit result, short-circuits
11	?:	ternary, right-assoc → mux
12	{} {{}}	concat / replicate (lowest)

```
// Bitwise vs Logical vs Reduction - the classic confusion
wire [3:0] a = 4'b1010, b = 4'b0101;
a &  b   // 4'b0000  bitwise  : vector in -> vector out
a && b   // 1'b1     logical  : "both non-zero?" -> 1 bit
&a       // 1'b0     reduction AND  (1&0&1&0)
|a       // 1'b1     reduction OR   == "a != 0"
^a       // 1'b0     reduction XOR  == parity (1 if ODD # of ones)
~&a      // 1'b1     NAND      ~|a -> 1'b0  NOR == "a == 0"
!a       // 1'b0     logical NOT    ~a -> 4'b0101  bitwise NOT
```

Operator	With `a=4'b10x1, b=4'b10x1`	Synth?
== !=	1'bx — any x/z poisons the result	yes
=== !==	1'b1 — x and z compared literally	**no**
==? !=?	x/z in **RHS only** = wildcard don't-care	yes (SV)

Use in RTL: `==` . **Use in testbench:** `===` so you can catch an `x` . A comparison against `x` with `==` is never true *and* never false — it is `x` , which an `if` treats as false.

1.7 Expression Width & Sign Rules H

Verilog resizes every operand of an expression to the width of the **widest operand, including the left-hand side** (self-determined operands are the exception). This single rule causes most silent RTL bugs.

Context	Rule
Context-determined	`+ - * / % & | ^ ~ ?: <<<` and the RHS of an assignment — all extended to max(LHS, operands)
Self-determined	shift **amount**, `{}` concat/replicate operands, `**` RHS, conditional of `?:`, all operands of `&& || ! == <` (result is 1 bit but operands size to each other)
Signed result	only if **every** operand is signed. One unsigned operand → the **whole expression** becomes unsigned

```
reg signed [7:0] a = -8'sd5;
reg        [7:0] b =  8'd2;
reg signed [8:0] r;
r = a + b;              // BUG: b unsigned -> whole expr unsigned,
                        //      a zero-extended -> 251+2 = 253
r = a + $signed(b);     // FIX
r = a + $signed({1'b0,b});   // FIX: widen, then sign

reg [3:0] x = 4'hF;  reg [4:0] y;
y = x + 1'b1;           // 5'b10000 = 16  (LHS is 5b -> ops widen to 5b)
y = (x + 1'b1) >> 1;    // 5'b01000 = 8   (parens still 5-bit here)
y = {1'b0,x} + 1'b1;    // 16 - explicit and portable

reg [7:0] p; reg [3:0] m=4'd15,n=4'd15;
p = m * n;              // 225 OK (LHS 8b widens the multiply)
p = (m * n) >> 4;       // 14  OK
$display("%0d", m*n);   // 225? NO -> self-determined 4b -> 4'd1 !!
```

Golden width rules: unsigned *N*+*N* needs **N+1** bits · *M*×*N* needs **M+N** bits · sum of *K* N-bit numbers needs **N+⌈log₂K⌉** bits · signed *M*×signed *N* needs M+N · signed *M*×unsigned *N* needs **M+N+1** (zero-extend the unsigned operand by one bit first).

1.8 Procedural Blocks & Assignment Forms

```
assign y = a & b;              // continuous - drives a NET, always live
always @(*)           ...      // comb, V-2001 auto sensitivity
always @(posedge clk)...       // sequential (edge)
always @(a or b)      ...      // explicit list (V-1995) - error prone
always_comb           ...      // SV: comb, checked
always_ff  @(posedge clk) ...  // SV: sequential, checked
always_latch          ...      // SV: intentional latch, checked
initial               ...      // TB only, runs once at t=0
final                 ...      // SV: runs once at end of sim
always begin ... end           // no sensitivity -> free-running (TB clk)
```

Feature	Blocking `=`	Non-blocking `<=`
Timing	RHS evaluated **and** LHS updated immediately, in the Active region	RHS sampled in Active region; LHS updated later in the **NBA** region
Flow	blocks the following statements	does not block — all updates appear parallel
Hardware	combinational (wires/muxes)	sequential (D flip-flops)
Use in	`always_comb` / `always @*`	`always_ff` / `@(posedge clk)`
Risk	read-after-write races across blocks	none — that is the point

Cliff Cummings' 8 coding guidelines (memorise — asked verbatim): ① sequential logic → `<=` ; ② latch → `<=` ; ③ combinational in an `always` → `=` ; ④ mixed seq+comb in one block → `<=` ; ⑤ never mix `=` and `<=` on the same variable; ⑥ never assign the same variable from more than one `always` block; ⑦ use `$strobe` not `$display` to print NBA-updated values; ⑧ never use `#0` delays.

1.9 Control Flow

```
if (sel) y = a; else y = b;          // priority -> mux / priority enc.

case (op)                            // exact 4-state match, no priority
  3'b000 : y = a + b;
  3'b001,
  3'b010 : y = a - b;                // multiple labels
  default: y = 8'h00;                // ALWAYS include default
endcase

casez (addr)                         // z and ? are don't-care  <-- OK
  8'b1???_???? : sel = 3'd0;         // priority decoder idiom
  8'b01??_???? : sel = 3'd1;
  default      : sel = 3'd7;
endcase
casex (addr)                         // x AND z don't-care  <-- BANNED

// SV modifiers - assertions, not synthesis pragmas
unique   case(...)  // exactly one match must exist (parallel + full)
unique0  case(...)  // at most one match (parallel, not full)
priority case(...)  // at least one match, order matters (full)

// Loops
for (i = 0; i < 8; i = i + 1)  ...   // synthesisable if bounds const
for (int i = 0; i < 8; i++)    ...   // SV: local loop var (safer)
while (cond) ... ;  repeat (8) ... ;  forever ... ;  do ... while(c);
foreach (mem[i]) mem[i] = '0;        // SV
```

Why `casex` is banned in RTL: if an uninitialised or bug-injected `x` reaches the selector, `casex` matches the **first** branch containing a don't-care — hiding the failure and producing a simulation/gate mismatch. Use `casez` (only `?`) or, in SV, `case ... inside` .

1.10 `generate` / `genvar`

```
genvar i;
generate
  for (i = 0; i < N; i = i + 1) begin : g_bit   // LABEL IS MANDATORY
     full_adder u_fa (.a(a[i]), .b(b[i]), .cin(c[i]),
                      .s(sum[i]), .cout(c[i+1]));
  end
  if (MODE == "FAST") begin : g_fast   cla u(...);  end
  else                begin : g_slow   rca u(...);  end
  case (WIDTH)
     8 : begin : g8  mul8  u(...); end
     default: begin : gN  mulN u(...); end
  endcase
endgenerate
// Hierarchical name of instance i: top.dut.g_bit[3].u_fa
```

- Elaborated at **compile time** — it replicates hardware, it is not a runtime loop.
- `genvar` is an elaboration-time integer; it does not exist in the netlist.
- Only **constant** expressions (params, localparams, genvars) allowed in the condition.
- Named blocks give unique, predictable instance names for constraints and waveform navigation.

1.11 Functions vs Tasks

Feature	`function`	`task`
Simulation time	zero — no `#`, `@`, `wait`	may consume time
Returns	exactly one value (via its name, or `return`)	none; use `output`/`inout` args
Arguments	≥1 input, no output (SV allows `output`/`ref`)	any number, any direction
Calls	can call functions, **not** tasks	can call both
Used as	an expression operand	a standalone statement
Synthesis	yes (inlined combinational logic)	rarely — TB use

```
function automatic [3:0] onehot2bin (input [15:0] oh);
   integer k;
   begin
      onehot2bin = 4'd0;
      for (k = 0; k < 16; k = k + 1)
         if (oh[k]) onehot2bin = k[3:0];
   end
endfunction

function automatic int clog2 (input int v);   // SV has $clog2()
   begin  clog2 = 0;
      while ((1 << clog2) < v) clog2++;  end
endfunction

task automatic drive_beat (input [31:0] d);   // automatic => re-entrant
   begin @(posedge clk); data <= d; valid <= 1'b1;
         @(posedge clk); valid <= 1'b0; end
endtask
```

`static` **vs** `automatic` H: Verilog defaults to **static** — one shared copy of the locals. Two concurrent calls (e.g. from a `fork`) corrupt each other, and recursion is impossible. `automatic` allocates locals on the stack per call → re-entrant, recursive, thread-safe. Always write `task automatic` / `function automatic` in testbenches. In SV, everything inside a `class` , and any `module` declared `module automatic` , is automatic by default.

1.12 Parameters, Constants & Directives

```
parameter  W    = 8;          // overridable per instance
localparam MAX  = (1<<W)-1;   // derived, NOT overridable  <-- prefer
parameter type T = logic [7:0];  // SV: type parameter
specparam  tPD  = 1.2;        // timing only, inside specify

`define WIDTH 8               // text macro - GLOBAL, no scope!
`ifdef SIM ... `elsif FPGA ... `else ... `endif
`ifndef GUARD_V
`define GUARD_V   ... `endif  // include guard
`include "defs.vh"
`timescale 1ns/1ps            // unit / precision
`default_nettype none         // top of file  <-- ALWAYS
`default_nettype wire         // bottom of file
`undef  `line  `resetall  `celldefine  `unconnected_drive

// Override styles
mod #(.W(16)) u1 (...);       // instance param  <-- USE THIS
defparam u1.W = 16;           // DEPRECATED, breaks tools
```

	`parameter`	`localparam`	`` `define ``
Scope	module	module	**global** from point of definition
Overridable	yes	no	no (but `` `undef ``+redefine)
Typed	yes	yes	no — raw text substitution

1.13 System Tasks & Format Specifiers

Task	When it prints
`$display`	immediately, in the Active region, with newline
`$write`	same, no newline
`$strobe`	end of timestep (**Postponed**) — sees final NBA values
`$monitor`	Postponed region of every timestep in which any argument changed (only one active at a time; `$monitoron/off`)

```
%b %o %d %h  %0d (no pad)  %s %c %e %f %g %t %m (hier name) %v (strength)
%p (SV pretty-print struct/array)   %%  \n \t \\ \"
$time $stime $realtime $timeformat(-9,2," ns",10)
$random(seed) $urandom $urandom_range(hi,lo) $dist_uniform
$finish $stop $exit $fatal $error $warning $info
$readmemh("init.hex", mem, start, end)   $readmemb  $writememh
$fopen $fdisplay $fwrite $fclose $fgets $fscanf $sformatf $sscanf
$dumpfile("w.vcd") $dumpvars(0, tb)  $dumpon $dumpoff  // VCD
$signed() $unsigned() $clog2() $bits() $countones() $onehot() $onehot0()
$isunknown() $past() $rose() $fell() $stable() $changed()  $assertoff
$test$plusargs("FOO")  $value$plusargs("SEED=%d", s)
```

// §2 SIMULATION SEMANTICS — THE EVENT QUEUE

2.1 IEEE 1364 Stratified Event Queue (per timestep)

#	Region	What happens here
1	**Active**	blocking assignments; evaluate RHS of `<=`; continuous assigns; primitive evaluation; `$display`/`$write`; execute `always`/`initial` bodies. Order **within** this region is arbitrary → races.
2	Inactive	`#0` blocking assignments (never use these)
3	**NBA**	update the LHS of all non-blocking assignments scheduled in ①. This is what makes flops "sample old values"
4	Monitor / Postponed	`$strobe`, `$monitor` — read-only, values are final

SystemVerilog (1800) refines this into 17 regions; the four that get asked about are **Preponed** (sample values for assertions/clocking-block inputs, before anything changes) → Active → **Observed** (concurrent assertion evaluation) → **Reactive** (`program` blocks, clocking-block output drives). This split is *the* reason a clocking block removes TB↔DUT races.

2.2 Delay Forms — Inter vs Intra-Assignment

```
#5  a = b;   // wait 5, THEN sample b, then assign     (inter)
a = #5  b;   // sample b NOW, wait 5, then assign a    (intra)
a <= #5 b;   // sample b NOW, schedule a for t+5       (transport delay)
@(posedge clk) a = b;      // wait for edge, then assign
a = @(posedge clk) b;      // sample b now, assign at next edge
wait (en) a = b;           // level-sensitive wait
```

Why `a <= #1 b;` in gate-level TBs: the intra-assignment delay guarantees the driver changes *after* the clock edge, modelling a real clock-to-Q and dodging hold-time races in back-annotated simulation.

2.3 The Race-Condition Catalogue H

Race	Cause & fix
Two `always` blocks, `=` on shared signal	Active-region order is arbitrary → shift register or transparent path, at random. **Fix:** use `<=`.
TB drives with `=` on the same edge the DUT samples	DUT may see old or new data. **Fix:** drive with `<=`, or a clocking block with output skew.
`$display` of an NBA target	prints the pre-update value. **Fix:** `$strobe`.
Same variable from two blocks	illegal for synthesis, non-deterministic in sim.
`#0` to "fix ordering"	only moves the race to the Inactive region. **Never use.**
Clock and data generated in the same `initial`	zero skew ambiguity — generate the clock in its own `always`.

2.4 `` `timescale ``

```
`timescale 1ns / 100ps    // time_unit / time_precision
// #1.06  ->  rounds to 1.1 ns (precision = 100 ps)
```

- Precision must be ≤ unit. The **smallest precision in the whole design** sets the simulator's tick, which slows everything down.
- A file with no `` `timescale `` inherits the previous compiled file's — a real, order-dependent bug. Use `timeunit` / `timeprecision` inside SV modules instead.

// §3 SYNTHESIS — RTL TO GATES

3.1 Synthesisable vs Not

Synthesisable	NOT synthesisable
`assign`, `always`, `always_comb`/`ff`/`latch`	`initial` (except FPGA RAM/reg init), `final`
`if`, `case`, `casez`, `for` (const bounds)	`casex` (allowed but banned by rule), `while`/`forever` with data-dependent exit
`function`, `task` (no timing)	delays `#`, `wait`, `fork`/`join`
`generate`, `parameter`, `localparam`	`===`, `!==`, `real`, `time`, `defparam`
`+ - * & | ^ ~ << >> ?: {}`, compare	`/` and `%` by a non-constant, non-power-of-2
memory arrays → RAM/ROM inference	`force`/`release`, `$display`, classes, mailboxes, randomize
tri-state via `1'bz` (I/O pads only)	UDPs, `specify` blocks, hierarchical references

3.2 Inference Recipes — memorise these

```
// D flip-flop ------------------------------------------------
always_ff @(posedge clk) q <= d;

// DFF + ASYNC active-low reset  (reset in sensitivity list)
always_ff @(posedge clk or negedge rst_n)
   if (!rst_n) q <= '0;  else q <= d;

// DFF + SYNC reset  (reset NOT in sensitivity list)
always_ff @(posedge clk)
   if (!rst_n) q <= '0;  else q <= d;

// DFF + clock enable  -> maps to CE pin, NOT a mux (usually)
always_ff @(posedge clk)
   if (!rst_n) q <= '0;  else if (en) q <= d;

// 4:1 MUX  (full case -> parallel, no priority)
always_comb
   case (sel)
     2'd0: y = a;  2'd1: y = b;
     2'd2: y = c;  default: y = d;
   endcase
assign y = sel[1] ? (sel[0]?d:c) : (sel[0]?b:a);   // same gates

// PRIORITY encoder (if/else if chain = priority)
always_comb begin
   y = 3'd7;                                  // default first!
   if      (req[0]) y = 3'd0;
   else if (req[1]) y = 3'd1;
   else if (req[2]) y = 3'd2;
end

// LATCH (deliberate)
always_latch
   if (en) q <= d;

// SYNC ROM
always_ff @(posedge clk)
  case (addr)
    0: d <= 8'h3C; ...
  endcase

// TRI-STATE (I/O pad only)
assign pad = oe ? dout : 1'bz;
assign din = pad;

// SYNC single-port RAM
always_ff @(posedge clk) begin
   if (we) mem[addr] <= din;
   dout <= mem[addr];   // READ-OLD
end
```

RAM read style	Code inside `always_ff`	Result
Read-first (old data)	`if(we) mem[a]<=d;` then `dout<=mem[a];`	returns the value **before** the write
Write-first (new data)	`if(we) begin mem[a]<=d; dout<=d; end else dout<=mem[a];`	write-through / read-during-write bypass
No-change	`if(we) mem[a]<=d; else dout<=mem[a];`	output holds during a write (lowest power)

3.3 Latch Inference — cause & cure

A latch appears whenever a variable in a **combinational** block is **not assigned on every possible path**, so the tool must remember the old value.

```
// THREE latch bugs                    // THREE cures
always @(*) begin                      always_comb begin
  if (sel) y = a;      // no else         y = 1'b0;      // 1. default
end                                      x = a;         //    assign all
always @(*) begin                        case (st)      // 2. full case
  case (st)                                2'd0: y = a;
    2'd0: y = a;       // no default       default: y = b;  // 3. default
  endcase                                endcase
end                                    end
```

Feature	Latch (level-sensitive)	Flip-flop (edge)
Trigger	transparent while enable is active	samples only on the clock edge
Inferred by	incomplete `if`/`case` in comb block	`posedge`/`negedge` in sensitivity list
Timing (STA)	hard — time borrowing, transparency windows	simple setup/hold, deterministic
Area / power	~½ the transistors; passes input glitches	bigger; immune to glitches between edges
Test (DFT)	poor scan coverage	full scan support

When a latch is legitimate: inside an integrated clock-gating cell, in low-power datapaths with time borrowing, and in some analog/mixed-signal handshakes. Everywhere else it is a bug — declare it with `always_latch` so the tool knows you meant it.

3.4 `full_case` / `parallel_case` — why they are banned H

Pragma	Tells synthesis	Why it bites
`full_case`	"all unlisted selector values are impossible → treat as don't-care"	Simulation holds the old value (latch) or takes `default`; the gates output an optimised don't-care. **Sim ≠ synth**, and the mismatch usually only shows up in gate-level sim.

Pragma	Tells synthesis	Why it bites
`parallel_case`	"branches are mutually exclusive → build a parallel mux, no priority"	If the branches actually overlap, RTL uses priority order but the gates arbitrate differently → silent functional bug.

Do this instead: write a real `default` for full coverage, and use SV `unique case` / `priority case`. Those are *assertions* that fire a runtime error when violated **and** convey the same synthesis intent — so simulation and gates stay consistent.

3.5 Synthesis Golden Rules

- One `always` block drives one set of signals. Never drive a signal from two blocks.
- `<=` for sequential, `=` for combinational. Never mix on one variable.
- Never put a `#delay` in RTL — synthesis ignores it and sim then differs.
- Complete sensitivity lists: use `always_comb` (or `@*`), never a hand-written list.
- Assign every output on every path, or you get a latch.
- Don't gate clocks with random logic — use an ICG cell.
- Register all module outputs where you can: it makes timing budgets composable.
- No combinational loops, no `initial` in ASIC RTL, no hierarchical references.
- Reset only what must be reset — datapath pipeline registers usually need no reset (saves area and reset-tree routing).

// §4 COMBINATIONAL BUILDING BLOCKS

4.1 Bit-Twiddling Idioms Interviewers Love

Goal	Expression
Isolate lowest set bit	`x & (~x + 1)` i.e. `x & -x`
Clear lowest set bit	`x & (x - 1)`
Is power of two (and ≠0)	`x && !(x & (x-1))` or `$onehot(x)`
Is zero / is non-zero	`~|x` / `|x`
All ones	`&x`
Odd parity	`^x`
Count ones (popcount)	`$countones(x)`; RTL = adder tree
Binary → Gray	`g = b ^ (b >> 1)`
Gray → binary	`b[i] = ^g[MSB:i]` (XOR prefix chain)
Two's complement negate	`~x + 1'b1`
Sign-extend N→M	`{{(M-N){x[N-1]}}, x}`
Swap without temp	`{a,b} <= {b,a};`
Reverse bits	`{<<{x}}` (SV streaming operator)
Saturating add	`s = a+b; y = s[N] ? {N{1'b1}} : s[N-1:0];`
Round-half-up (drop k bits)	`(x + (1<<(k-1))) >> k`

4.2 Adder Architectures M

Adder	Delay	Area	Use
Ripple-carry (RCA)	O(N)	O(N)	small, slow, lowest power
Carry-lookahead (CLA)	O(log N)	O(N log N)	classic fast adder
Carry-select	O(√N)	~2×	speed/area compromise
Carry-skip	O(√N)	O(N)	cheap speed-up
Kogge-Stone (prefix)	O(log N), min depth	large, heavy wiring	max speed, wide adders
Brent-Kung	O(log N), 2× depth	fewer cells	area-efficient prefix
Carry-save (CSA)	O(1) per stage	O(N)	multi-operand / multipliers

Generate/propagate: `g=a&b`, `p=a^b`, `c[i+1]=g[i] | (p[i] & c[i])`, `sum=p^c`. In practice: just write `assign sum = a + b;` and let the synthesiser pick from its datapath library based on your timing constraint — say this in an interview.

4.3 Fast Building Blocks

```
// One-hot priority arbiter, fixed priority, LSB wins ---------
assign gnt = req & (~req + 1'b1);            // O(N) carry chain

// Same, explicit (shows the priority chain)
assign gnt[0] = req[0];
generate for (i=1;i<N;i=i+1) begin: g_arb
   assign gnt[i] = req[i] & ~(|req[i-1:0]);
end endgenerate

// ROUND-ROBIN arbiter (mask + rotate) ----------------------
wire [N-1:0] masked   = req & mask;                // mask = higher-
wire [N-1:0] gnt_hi   = masked & (~masked + 1'b1); //  priority-than-last
wire [N-1:0] gnt_lo   = req    & (~req    + 1'b1);
assign gnt = |masked ? gnt_hi : gnt_lo;            // wrap around
always_ff @(posedge clk) if (|gnt) mask <= ~((gnt - 1) | gnt);

// Barrel shifter (log N stages, no priority chain) ----------
wire [31:0] s0 = sh[0] ? {d[30:0],1'b0} : d;
wire [31:0] s1 = sh[1] ? {s0[29:0],2'b0}: s0;
wire [31:0] s2 = sh[2] ? {s1[27:0],4'b0}: s1;   // ... 5 stages

// Binary <-> Gray -------------------------------------------
assign gray = bin ^ (bin >> 1);
generate for (i=0;i<N;i=i+1) begin: g_g2b
   assign bin[i] = ^gray[N-1:i];
end endgenerate

// Edge detector (needs a clock - see 5.1) -------------------
always_ff @(posedge clk) d_q <= d;
assign rise = d & ~d_q;   assign fall = ~d & d_q;
assign any  = d ^ d_q;
```

// §5 SEQUENTIAL LOGIC & FSMS

5.1 Counters, Shift Registers, LFSR

```
// Mod-N counter with tick ----------------------------------
always_ff @(posedge clk or negedge rst_n)
  if (!rst_n)           cnt <= '0;
  else if (cnt == N-1)  cnt <= '0;
  else                  cnt <= cnt + 1'b1;
assign tick = (cnt == N-1);          // 1-cycle pulse every N

// Up/down/load counter
always_ff @(posedge clk)
  if      (load) cnt <= din;
  else if (en)   cnt <= up ? cnt + 1'b1 : cnt - 1'b1;

// SISO/PISO shift register  (NBA gives real stages)
always_ff @(posedge clk)
  if (ld) sr <= din; else sr <= {sr[N-2:0], si};   // shift left
// arithmetic right shift keeps the sign
always_ff @(posedge clk) sr <= {sr[N-1], sr[N-1:1]};

// Ring counter (N states, N FF)   Johnson/twisted (2N states)
sr <= {sr[N-2:0], sr[N-1]};        sr <= {sr[N-2:0], ~sr[N-1]};

// Maximal-length LFSR: 2^N-1 states, never all-zero (Fibonacci)
always_ff @(posedge clk or negedge rst_n)
  if (!rst_n) lfsr <= 8'hFF;                    // seed != 0
  else        lfsr <= {lfsr[6:0], lfsr[7]^lfsr[5]^lfsr[4]^lfsr[3]};
// taps: 8->[8,6,5,4]  16->[16,15,13,4]  32->[32,22,2,1]
```

Uses of an LFSR: pseudo-random stimulus, BIST pattern generation, CRC, scrambling, and as a **cheap fast counter** — an LFSR has no carry chain, so it beats a binary counter at high frequency when you only need "count to N", not the actual value.

5.2 Moore vs Mealy

Feature	Moore	Mealy
Output	Y = f(state)	Y = f(state, input)
Latency	reacts 1 cycle later	reacts in the same cycle
Glitches	glitch-free (state-decoded)	input glitches reach the output
States	usually more	usually fewer
Timing	output not in the input's critical path	combinational input→output path crosses the module boundary
Prefer when	you want registered, safe outputs	you must respond immediately

Best of both: compute Mealy next-state logic but **register the output** ("registered Mealy") — you get the low state count with a glitch-free, timing-clean output, at the cost of one cycle.

5.3 State Encoding

Encoding	FFs for N states	Next-state logic	Best for
Binary	$\lceil \log_2 N \rceil$	deep decode	area-critical ASIC, many states
Gray	$\lceil \log_2 N \rceil$	deep decode	low switching power, async pointers
One-hot	N	very shallow (1 AND/OR level)	FPGA (FFs are free), high f_{max}
One-cold	N	shallow	same, reset-to-all-ones tech
Johnson	N/2	shallow	compromise, glitch-free decode

5.4 Canonical 3-Block FSM Template

```
module fsm #(parameter W=8)
( input logic clk, rst_n, in_valid, output logic out_pulse );

  typedef enum logic [1:0] {IDLE=2'b00, RUN=2'b01, DONE=2'b10} state_e;
  state_e state, nxt;                    // enum: waves show names

  // 1) STATE REGISTER - sequential, nothing else
  always_ff @(posedge clk or negedge rst_n)
     if (!rst_n) state <= IDLE; else state <= nxt;

  // 2) NEXT-STATE - pure combinational
  always_comb begin
     nxt = state;                        // default = hold (no latch)
     unique case (state)
        IDLE: if (in_valid) nxt = RUN;
        RUN :               nxt = DONE;
        DONE:               nxt = IDLE;
        default:            nxt = IDLE;    // safe recovery from x
     endcase
  end

  // 3) OUTPUT - registered (Moore, glitch-free)
  always_ff @(posedge clk or negedge rst_n)
     if (!rst_n) out_pulse <= 1'b0;
     else        out_pulse <= (nxt == DONE);
endmodule
```

Why three blocks? ① the sequential block is trivially reviewable and always correct; ② the combinational block holds all the behaviour in one readable `case`; ③ output logic can be swapped between Moore/Mealy/registered without touching the rest. A 1-block FSM is compact but mixes concerns and easily produces unintended registered outputs; a 2-block FSM is a fine alternative.

5.5 Sequence Detector — `1011` M

State (bits seen)	in=`0` →	in=`1` →	Mealy out	Moore out
`S0` : none	`S0`	`S1`	`0` / `0`	`0`
`S1` : 1	`S2`	`S1`	`0` / `0`	`0`
`S2` : 10	`S0`	`S3`	`0` / `0`	`0`
`S3` : 101	`S2`	`S1`‡ / `S0`†	`0` / `1`	`0`
`S4` : 1011	Moore only — extra state that asserts the output		—	`1`

‡ **overlapping** detector: after a hit, the trailing `1` is reused as the start of the next match → go to `S1`. † **non-overlapping**: discard everything, restart at `S0`. Mealy needs 4 states, Moore needs 5. Always ask the interviewer which variant they want — that question alone scores points.

```
// Mealy, overlapping, one-liner style
always_comb begin
   nxt = st;  det = 1'b0;
   case (st)
     S0: nxt = din ? S1 : S0;
     S1: nxt = din ? S1 : S2;
     S2: nxt = din ? S3 : S0;
     S3: begin nxt = din ? S1 : S2;  det = din; end   // 1011 !
   endcase
end
```

5.6 Pipelining, Latency & Throughput

Term	Meaning
Latency	cycles from a given input to its own output
Throughput	results per cycle (II = initiation interval)
Pipelining	cut a long comb path with registers → higher f_{max}, **more** latency, same throughput/cycle
Retiming	tool moves existing registers across logic to balance stage delays; register count preserved
Register balancing	retiming driven by the timing constraint
Unrolling / parallelism	N copies → N× throughput, N× area
Resource sharing	time-multiplex one big unit → less area, less throughput
C-slow	interleave N independent streams through one pipeline

Adding a pipeline stage inside a **feedback loop** does not help — the loop's recurrence (its "iteration bound") sets the ceiling. That is why accumulators and FSM feedback paths are so often the critical path. Fixes: loop unrolling with a carry-save accumulator, or algorithmic transformation (look-ahead / retiming across the loop).

5.7 Valid/Ready Handshake (AXI-stream style)

```
// Rules: a transfer happens when (valid && ready) on a clock edge.
// 1. VALID must NOT wait for READY   (else deadlock)
// 2. READY may wait for VALID
// 3. Once VALID is high it must stay high, with stable data,
//    until the transfer completes.
assign xfer = valid && ready;

// Skid buffer control core (output mux omitted) ------------
always_ff @(posedge clk or negedge rst_n)
  if (!rst_n) begin  full <= 1'b0;  end
  else begin
    if (in_valid && in_ready && !(out_valid && out_ready) && out_valid)
        begin skid <= in_data;  full <= 1'b1;  end
    else if (out_valid && out_ready) full <= 1'b0;
  end
assign in_ready = !full;
```

// §6 CLOCKING, RESET & CLOCK-DOMAIN CROSSING

6.1 Metastability & MTBF H

If a flip-flop's D input changes inside the setup/hold aperture, the output can hang at an intermediate voltage for an unbounded time before resolving to 0 or 1 — **arbitrarily**, and possibly differently for each fanout load. It always resolves eventually; the question is only "how likely is it to still be undecided when the next stage samples it?"

$$MTBF = e^{(t_r/\tau)} \;/\; (C_1 \cdot f_{clk} \cdot f_{data})$$

Term	Meaning	How to improve MTBF
t_r	settling time available = $T_{clk} - t_{setup} - t_{cq} - t_{comb}$	**add a sync stage** (adds a whole T_{clk}) — exponential gain
τ	flop's metastability resolution time constant	use faster/dedicated sync flops
C_1	aperture-width constant of the flop	library choice
f_{clk}	destination clock frequency	lower it (rarely an option)
f_{data}	toggle rate of the async input	filter/qualify the input

Key insight: MTBF improves **exponentially** with settling time but only **linearly** with the frequencies. That is why 2 flops usually suffice, 3 are used above ~500 MHz or for safety-critical logic, and why you must never put combinational logic *between* synchroniser stages — it eats t_r.

6.2 CDC Technique Selection

What is crossing	Correct technique
1-bit level / control	2-FF (dual-rank) synchroniser
1-bit pulse, **fast → slow**	**Toggle** synchroniser (pulse→level→2FF→edge-detect), or stretch/handshake. A plain 2-FF **will drop the pulse**.
1-bit pulse, slow → fast	2-FF + edge detect is fine (pulse is ≥ 1 dest cycle wide)
Multi-bit data, low rate	MUX-recirculation / DMUX: sync a single `valid` bit, let the data go combinationally and capture it with the synced enable
Multi-bit data, streaming	**Async FIFO** with Gray-coded pointers
Multi-bit, must be atomic	4-phase req/ack **handshake** (slowest, safest)
Bus of related counters	Gray code the counter, then 2-FF the Gray bits
Reset	Reset synchroniser (async assert, sync de-assert)

The classic multi-bit blunder: running a binary bus through parallel 2-FF synchronisers. Each bit resolves independently, so on the transition `0111 → 1000` the destination can latch **any** intermediate value (e.g. `1111` or `0000`). Gray code fixes this because only one bit changes per increment — the worst case is simply seeing the old or the new value, both of which are legal.

6.3 The Synchroniser Toolbox

```
// ---- 2-FF synchroniser (1-bit level) ----------------------
(* ASYNC_REG = "TRUE" *) reg meta_q, sync_q;      // Xilinx attribute
always_ff @(posedge clk_dst or negedge rst_dst_n)
  if (!rst_dst_n) {sync_q, meta_q} <= 2'b00;
  else            {sync_q, meta_q} <= {meta_q, async_in};
assign sync_out = sync_q;

// ---- TOGGLE pulse synchroniser (fast -> slow) --------------
always_ff @(posedge clk_src)                  // 1. pulse -> level
  if (pulse_in) tog <= ~tog;
always_ff @(posedge clk_dst)                  // 2. 2-FF sync
  {s2,s1,s0} <= {s1,s0,tog};
assign pulse_out = s2 ^ s1;                   // 3. edge detect

// ---- 4-phase handshake (safest, ~5 cycles per word) --------
// src: data valid -> req=1 ... wait ack=1 -> req=0 ... wait ack=0
// dst: sees req -> capture data -> ack=1 ... sees req=0 -> ack=0

// ---- Reset synchroniser (async assert, SYNC de-assert) -----
always_ff @(posedge clk or negedge arst_n)
  if (!arst_n) {rst_n, r1} <= 2'b00;          // assert immediately
  else         {rst_n, r1} <= {r1, 1'b1};     // release on clk edges
```

6.4 Asynchronous FIFO H

```
// Depth 16 -> 4 address bits + 1 extra MSB = 5-bit pointers
wptr_bin  <= wptr_bin + (wr_en & ~full);
wptr_gray <= (wptr_bin+1) ^ ((wptr_bin+1) >> 1);  // bin -> gray
// each gray pointer crosses via its own 2-FF synchroniser

// EMPTY  : read side sees write pointer catch up  -> ALL bits equal
assign empty = (rptr_gray == wptr_gray_sync);
// FULL   : write side sees read pointer 1 lap behind ->
//          top TWO bits inverted, remaining bits equal
assign full  = (wptr_gray == {~rptr_gray_sync[4:3],
                              rptr_gray_sync[2:0]});
```

Question	Answer
Why Gray code the pointers?	Only one bit changes per increment, so a partially-synchronised sample is always either the old or the new pointer — never a bogus intermediate.
Why N+1 bits for a 2^N-deep FIFO?	To distinguish "pointers equal because empty" from "equal because full, one lap apart". The extra MSB is the wrap bit.
Why the **two** MSBs inverted for full?	In Gray code, "one lap ahead" flips both of the top two bits, not just the MSB (Gray is a reflected code).
Are the flags exact?	They are **pessimistic and safe**: `full` may assert a bit early and `empty` a bit late, because each side sees a delayed copy of the other pointer. They are never optimistic — so you can never overflow or underflow.
Compare which pointer?	The **registered** Gray pointer. Deriving `full` from the *next* Gray value closes a combinational loop (`full` → `wbin_next` → `wgray_next` → `full`) that hangs simulation. Compare the registered pointer, or register the flag itself.
Which side computes which flag?	`full` in the write domain, `empty` in the read domain — always computed locally, using a synchronised copy of the remote pointer.
Minimum depth?	Sized from burst length and the rate mismatch: depth ≥ burst × (1 − f_{slow}/f_{fast}) plus sync latency margin (~2–3 words).

6.5 Reset Architecture

	Synchronous reset	Asynchronous reset
In sensitivity list?	no	yes (`or negedge rst_n`)
Needs a running clock?	**yes**	no — works before the PLL locks
Glitch immunity	high (clock filters glitches)	low — a glitch resets the chip
Timing cost	extra gate in the D path, lengthens the critical path	none in the data path; but needs recovery/removal checks
Reset tree	ordinary timed path	must be balanced like a clock tree
Risk	reset pulse narrower than a clock cycle is missed	metastability on **de-assertion**

Industry answer: use both. "Asynchronous assertion, synchronous de-assertion" via a reset synchroniser — you get a reset that works with no clock, and a release that is aligned to the clock so recovery/removal is met. Then hold reset for ≥ 2–3 destination clocks in every domain, and give each clock domain its own reset synchroniser fed from the same global reset.

Check	Definition
Recovery time	minimum time the async reset must be **de-asserted before** the next active clock edge (the "setup" of reset release)
Removal time	minimum time the async reset must stay asserted **after** the clock edge (the "hold" of reset release)

6.6 Clock Gating, Division & Muxing

```
// GLITCHY - never do this in RTL
assign gclk = clk & en;      // en changing while clk=1 -> runt pulse

// Correct ICG: latch the enable on the LOW phase of clk
always_latch if (!clk) en_l <= en;      // negative level-sensitive
assign gclk = clk & en_l;               // en_l is stable while clk=1
// In practice instantiate the library cell: CKLNQD1 / BUFGCE / ICG

// Divide by 2 (50% duty, from one FF)
always_ff @(posedge clk) clk_div2 <= ~clk_div2;

// Divide by even N: toggle when counter hits N/2-1
// Divide by ODD N with 50% duty: build two dividers, one on
// posedge and one on negedge, then OR (or XOR) them together
always_ff @(posedge clk) if (c==N-1) c<=0; else c<=c+1;
always_ff @(posedge clk) p <= (c < (N-1)/2);
always_ff @(negedge clk) n <= (c < (N-1)/2);
assign clk_odd = p | n;        // 50% duty for odd N

// SAFEST divider for synthesis: don't make a clock at all -
// make a clock ENABLE and keep one clock domain.
assign en_div = (c == N-1);
always_ff @(posedge clk) if (en_div) data <= next;
```

Never build a clock out of combinational logic or a ripple counter's Q output. Generated clocks are unbalanced, un-analysable in STA, break DFT scan, and add skew. Use a clock enable, an ICG cell, or a PLL/MMCM/clock divider hard macro. For run-time clock **selection**, use a glitch-free clock mux (each branch: sync the select into that domain, gate off before switching, then gate on) — never a bare `?:`.

// §7 STATIC TIMING ANALYSIS ESSENTIALS

7.1 The Two Equations H

Check	Inequality
Setup (max delay, next edge)	$t_{cq} + t_{comb,max} + t_{setup} \le T_{clk} + t_{skew} - t_{jitter}$
Hold (min delay, same edge)	$t_{cq,min} + t_{comb,min} \ge t_{hold} + t_{skew}$

$t_{skew} = t_{clk,capture} - t_{clk,launch}$. **Positive skew** (capture clock late) *helps setup, hurts hold.* **Negative skew** does the opposite. Slack = required − arrival; negative slack = violation.

$$f_{max} = 1 / (t_{cq} + t_{comb,max} + t_{setup} - t_{skew})$$

	Setup violation fixes	Hold violation fixes
RTL	pipeline/retime, shorten logic depth, better encoding, duplicate high-fanout drivers	rarely an RTL issue
Synthesis / P&R	upsize cells, better arch from datapath library, useful skew, restructure	**insert buffers/delay cells** on the short path, downsize, reroute
System	lower the clock frequency	frequency does **not** help — hold is frequency-independent

Interview favourite: "Can you fix a hold violation by slowing the clock?" **No.** Hold is checked between the launch and capture edges of the **same** clock event — the period cancels out of the inequality. Only adding delay to the data path (or reducing clock skew) fixes hold. Conversely, a setup violation *can* always be fixed by slowing the clock.

7.2 Timing Exceptions & Constraints (SDC)

```
create_clock -name clk -period 5.0 [get_ports clk]      # 200 MHz
create_generated_clock -divide_by 2 -source [get_ports clk] ...
set_input_delay  -clock clk 1.2 [get_ports din]
set_output_delay -clock clk 1.0 [get_ports dout]
set_clock_uncertainty 0.15 [get_clocks clk]             # jitter+margin
set_false_path   -from [get_clocks clk_a] -to [get_clocks clk_b]
set_max_delay    -datapath_only 4.0 -from ... -to ...   # CDC bus
set_multicycle_path 2 -setup -from [get_pins ...]
set_multicycle_path 1 -hold  -from [get_pins ...]       # ALWAYS pair
set_case_analysis 0 [get_ports test_mode]
```

Term	Meaning
False path	a topological path that can never be exercised → don't analyse it (e.g. across a synchroniser, or test-only logic)
Multicycle path	logic legitimately given N cycles to settle; you must set the **hold** exception too or you create hold violations
Jitter / uncertainty	cycle-to-cycle clock variation, budgeted out of the period
OCV / derating / AOCV	margin for on-chip process/voltage/temperature variation between launch and capture paths
PVT corner	slow-slow/low-V/high-T worst-case for setup; fast-fast/high-V/low-T worst-case for hold
Clock latency	source (before the port) + network (insertion delay through the tree)

// §8 SYSTEMVERILOG FOR DESIGN

8.1 What SV Adds Over Verilog (design side)

Verilog pain	SystemVerilog fix
`reg` vs `wire` confusion	`logic` — one type, single-driver checked
`always` intent is implicit	`always_comb` / `always_ff` / `always_latch` — tool checks you
state encoded as raw bits	`enum` — named states in waveforms, type-checked
huge port lists	`interface` + `modport`
related signals passed separately	`struct packed` — one bus, named fields
constants duplicated per file	`package` + `import`
`full_case/parallel_case` pragmas	`unique/priority case` (checked at run time)
TB races	`clocking` blocks, `program` blocks

`always_comb` **vs** `always @*` **— three real differences:** ① `always_comb` executes **once at time 0** so outputs are initialised (`@*` waits for a change); ② it is sensitive to signals read inside **functions called from the block**, which `@*` misses; ③ it makes the variables it writes illegal to drive from anywhere else — a compile-time single-driver check. It also warns if the block infers a latch.

8.2 Types, Enums, Structs, Packages

```
typedef logic [31:0] word_t;
typedef enum logic [1:0] {IDLE=2'b00, BUSY=2'b01, DONE=2'b10} state_e;
state_e st;  st = BUSY;            // type-safe, shows names in waves
st.next()  st.prev()  st.name()  st.num()  st.first()  st.last()

typedef struct packed {            // PACKED = one contiguous vector,
   logic [3:0]  opcode;            // synthesisable, can go on a port
   logic [11:0] addr;              // MSB..LSB in declaration order
   logic        valid;
} instr_t;                         // $bits(instr_t) == 17
instr_t ir;  ir.addr = 12'h0AB;  bus = ir;   // whole-struct assign

typedef union packed { logic [31:0] w; logic [3:0][7:0] b; } u_t;

package cpu_pkg;
   localparam int XLEN = 32;
   typedef enum logic [2:0] {ADD,SUB,AND_,OR_,XOR_} alu_op_e;
   function automatic logic [XLEN-1:0] sext(input logic [11:0] i);
      sext = {{20{i[11]}}, i};
   endfunction
endpackage
import cpu_pkg::*;                 // or cpu_pkg::alu_op_e explicitly
```

8.3 Arrays & Their Methods

Kind	Declaration	Notes / synthesis
Packed	`logic [3:0][7:0] a;`	a vector; sliceable; **synthesisable**
Unpacked (fixed)	`logic [7:0] a [16];`	memory; **synthesisable**
Dynamic	`int a []; + a=new[N];`	TB only. `size()` `delete()`
Queue	`int q [$];` or `[$:255]`	TB only. `push_back/front`, `pop_back/front`, `insert`, `delete`, `size`
Associative	`int a [string]; a[int]`	TB only, sparse. `exists()` `first()` `next()` `num()` `delete()`

```
// Array manipulation methods (TB; `with` = iterator expression)
// note: Vivado xsim rejects these on queues / dynamic arrays
q.sum()  q.product()  q.min()  q.max()  q.and()  q.or()  q.xor()
q.find(x) with (x > 10)        q.find_index with (item == 5)
q.find_first  q.find_last  q.unique  q.unique_index
q.sort()  q.rsort()  q.reverse()  q.shuffle()
foreach (arr[i,j]) arr[i][j] = 0;

// Streaming (pack/unpack/bit-reverse)
byte b[4]; word_t w;
w = {>>{b}};      // pack, left-to-right   (b[0] -> MSB)
{<<{b}} = w;      // unpack, reversed
rev = {<<{data}};      // reverse ALL bits
revB= {<<byte{data}}; // reverse byte order (endian swap!)
```

8.4 Interfaces & Modports

```
interface axi_lite_if #(parameter AW=32) (input logic clk, rst_n);
   logic [AW-1:0] awaddr;  logic awvalid, awready;
   logic [31:0]   wdata;   logic wvalid,  wready;

   modport master (output awaddr, awvalid, wdata, wvalid,
                   input  awready, wready, clk, rst_n);
   modport slave  (input  awaddr, awvalid, wdata, wvalid,
                   output awready, wready, input clk, rst_n);
   clocking cb @(posedge clk);          // for the testbench
      default input #1step output #1ns;
      output awaddr, awvalid;  input awready;
   endclocking
   modport tb (clocking cb, input clk, rst_n);
endinterface

module dut (axi_lite_if.slave bus);         // one port, whole bus
   always_ff @(posedge bus.clk) ...
endmodule
axi_lite_if #(32) u_if (clk, rst_n);
dut u_dut (.bus(u_if));
```

Pros: one connection instead of 40 ports, direction errors caught by modport, protocol assertions and coverage live with the bus. **Cons:** some synthesis flows dislike them at the top level, and hierarchy debug gets slightly indirect. Common compromise: interfaces for verification/sub-blocks, flat ports at the chip boundary.

8.5 Misc Design Constructs

```
i++;  i--;  ++i;  a += b;  a <<= 2;  a &= mask;   // C-style ops
x inside {[0:9], 12, 15}                          // set membership
case (op) inside 4'b1?0?: ...  endcase            // wildcard case
logic [7:0] q [4] = '{default:0};                 // assignment pattern
instr_t i2 = '{opcode:4'h3, addr:12'h0, valid:1'b1};
int arr[4] = '{1,2,3,4};
'0 '1 'x 'z                                       // fill literals
do_thing : begin ... end                          // labelled block
break; continue; return;                          // loop control
let is_hit(a) = valid && (tag == a);              // SV let macro
```

// §9 SYSTEMVERILOG VERIFICATION

9.1 Testbench Architecture

```
                +-----------------------------------------+
   Test  --->   |  Sequencer -> Driver ->[IF]-> DUT       |
                |                          |              |
   Scoreboard <-+-- Monitor <--------------+              |
                |       +--> Coverage    +--> Assertions  |
                +-----------------------------------------+
Stimulus: directed -> constrained-random -> coverage-driven closure
```

9.2 Classes & OOP

```
class Packet;
   rand  bit [7:0] addr;
   rand  bit [7:0] data [];            // dynamic array
   randc bit [3:0] id;                 // cyclic: no repeat until all
   bit [7:0] crc;
   static int count = 0;               // shared by all instances

   constraint c_addr { addr inside {[8'h10:8'h7F]};
                       addr % 4 == 0;                  }
   constraint c_len  { data.size() inside {[1:16]};
                       solve addr before data;         }
   constraint c_dist { addr dist {8'h10:=50, [8'h20:8'h30]:/50}; }

   function new(bit [7:0] a = 0);  addr = a;  count++;  endfunction
   virtual function void print();  $display("%p", this); endfunction
   function void pre_randomize();  ... endfunction
   function void post_randomize();      // NOT ^{addr,data} - a
      crc = addr;                       // dynamic array is unpacked
      foreach (data[i]) crc ^= data[i]; // and illegal in a concat
   endfunction
endclass

Packet p = new();
if (!p.randomize()) $fatal("randomization failed");
p.randomize() with { addr == 8'h40; data.size() == 4; };  // inline
p.addr.rand_mode(0);              // disable randomising a field
p.c_addr.constraint_mode(0);      // disable a constraint block
Packet q = new p;                 // SHALLOW copy (handles shared!)
```

Concept	One-liner
Shallow vs deep copy	`new p` copies handles, not the objects they point to. Write a `copy()`/`clone()` method for a deep copy.
`virtual` method	enables polymorphism — the **object's** type decides which body runs, not the handle's.
Abstract / pure	`virtual class` cannot be instantiated; `pure virtual` method must be overridden.
Static vs automatic	`static` members are shared per class; class methods are automatic by default.
`this` / `super`	current object / the parent class's version.
Parameterised class	`class fifo #(type T = int);` — a template.
`randc` vs `rand`	`randc` exhausts the whole range before repeating; only for ≤16-bit fields.
`solve A before B`	steers the **distribution** only; it never changes the legal solution space.
`:=` vs `:/` in `dist`	`:=` gives that weight to **each** value in the range; `:/` divides the weight **across** the range.
Soft constraint	`soft x == 0;` — dropped automatically if it conflicts with a hard one.

9.3 Processes & Synchronisation

Construct	Behaviour
`fork ... join`	waits for **all** spawned threads
`fork ... join_any`	continues after the **first** thread finishes
`fork ... join_none`	continues **immediately**; children run when the parent blocks
`disable fork`	kills all children of the current thread
`wait fork`	blocks until all children are done
`mailbox`	typed FIFO between threads: `put`/`get` (blocking), `try_put`/`try_get`, `peek`, `num`. `new(0)` = unbounded.
`semaphore`	key bucket for shared resources: `new(n)`, `get(k)`, `put(k)`, `try_get(k)`
`event`	`->e` triggers, `@e` waits (edge — can be missed), `wait(e.triggered)` is safe within the timestep

The `join_none` loop-variable trap H: spawned threads do not run until the parent blocks, so they all read the loop variable's **final** value. Fix by declaring the variable inside the loop (`for (int i=0;...)` in SV gives each iteration its own copy) or by copying it into an `automatic` local inside the `fork` body.

9.4 Functional Coverage

```
covergroup cg_pkt @(posedge clk);
   option.per_instance = 1;   option.at_least = 2;
   cp_addr : coverpoint pkt.addr {
       bins low    = {[0:63]};
       bins mid[4] = {[64:191]};        // 4 automatic sub-bins
       bins hi     = {[192:255]};
       bins walk[] = {1,2,4,8,16,32,64,128};
       ignore_bins rsv = {[240:255]};
       illegal_bins bad = {0};
   }
   cp_kind : coverpoint pkt.kind;       // enum -> auto bins
   x_ak    : cross cp_addr, cp_kind;
   cp_tr   : coverpoint st { bins t = (IDLE => BUSY => DONE); }
endgroup
cg_pkt cg = new();  cg.sample();  $display("%0.1f", cg.get_coverage());
```

	Code coverage	Functional coverage
Source	automatic from the tool	hand-written from the spec
Metrics	line, toggle, branch, condition, FSM state/arc	covergroups, coverpoints, crosses, SVA `cover`
Answers	"did I execute this code?"	"did I test this **feature**?"
Blind to	missing features (unwritten code is 100% covered)	anything you forgot to write a bin for

100% code coverage with 0% functional coverage means nothing; you need both, plus assertions. That three-part answer is what interviewers are listening for.

9.5 SystemVerilog Assertions (SVA)

Kind	Where	Semantics
Immediate `assert(e) else $error;`	inside procedural code	evaluated like an `if`, at that instant; no clock
Concurrent `assert property(...)`	module/interface/program	clocked, samples in the **Preponed** region, evaluates in Observed
Deferred `assert #0 (e);`	procedural	immediate, but reported at end of timestep → no glitch false-fires

```
// ---- Operators ------------------------------------------
a |-> b        // OVERLAPPING implication: b checked in the SAME cycle
a |=> b        // NON-overlapping: b checked NEXT cycle (== |-> ##1)
##1  ##[1:3]  ##[1:$]  ##[*]     // delay / range / eventually
a ##1 b ##2 c                    // a sequence
s[*3]     s[*1:5]                // consecutive repetition
s[->3]                           // goto: 3rd occurrence, gaps allowed
s[=3]                            // non-consecutive repetition
a and b   a or b   a intersect b // both / either / both, same length
a throughout b                   // a holds for the whole of b
a within b                       // a occurs inside b
not a      first_match(a)
b until c      b until_with c    // c excluded / included
s_until  s_eventually            // "strong": must actually happen
disable iff (!rst_n)             // abort while reset is active

// ---- Sampled values + assertion control -----------------
$rose(x) $fell(x) $stable(x) $changed(x) $past(x, n)
$onehot(x) $onehot0(x) $countones(x) $isunknown(x) $assertoff

// ---- Full example ---------------------------------------
property p_req_ack;
   @(posedge clk) disable iff (!rst_n)
   $rose(req) |-> ##[1:5] ack;              // ack within 1..5 cycles
endproperty
a_req_ack : assert property (p_req_ack)
            else $error("no ack: t=%0t", $time);
c_req_ack : cover property (p_req_ack);     // did we ever exercise it?

// data stability while stalled
assert property (@(posedge clk) disable iff(!rst_n)
   (valid && !ready) |=> (valid && $stable(data)) );
// one-hot state at all times
assert property (@(posedge clk) $onehot(grant) or (grant == '0));
// no overflow
assert property (@(posedge clk) !(push && full));
```

Trap	Explanation
Vacuous pass	If the antecedent never occurs, `a |-> b` "passes" trivially. Always pair every `assert` with a `cover` on the antecedent to prove you actually exercised it.
Weak vs strong	`##[1:$] b` at end of simulation passes weakly. Use `s_eventually` / `strong(...)` if it truly must happen.
Sampling	Concurrent assertions sample in the Preponed region, i.e. values **just before** the clock edge — that is why an assertion sees the "old" value of a signal the clock edge is changing.
Tool support	Check your simulator actually evaluates what you wrote. Vivado `xsim` compiles the `until` family (`until`, `until_with`, `s_until`) but never reports a violation — the assertion can never fail. Always pair a new property with a deliberately-false twin to prove the checker is live.
Missing `disable iff`	fires spuriously during reset.

9.6 UVM in One Panel

Component	Role
`uvm_sequence_item`	one randomisable transaction
`uvm_sequence`	generates a stream of items (`body()`, `` `uvm_do ``)
`uvm_sequencer`	arbitrates sequences → driver
`uvm_driver`	item → pin wiggles on the virtual interface
`uvm_monitor`	pins → item, broadcast on an analysis port
`uvm_agent`	sequencer+driver+monitor (`UVM_ACTIVE/PASSIVE`)
`uvm_scoreboard`	compares DUT output against a reference model
`uvm_env` / `uvm_test`	assembles agents / selects the scenario

```
// Phases in order. All TOP-DOWN except connect_phase (bottom-up)
build_phase -> connect_phase -> end_of_elaboration -> start_of_simulation
 -> run_phase (TIME-CONSUMING, the only task) -> extract -> check
 -> report -> final_phase
// run_phase sub-phases: reset -> configure -> main -> shutdown

`uvm_component_utils(my_driver)      // register with the FACTORY
`uvm_object_utils(my_item)
uvm_config_db#(virtual my_if)::set(null,"*","vif",u_if);   // set
uvm_config_db#(virtual my_if)::get(this,"","vif",vif);     // get
my_item::type_id::set_type_override(my_err_item::get_type()); // factory
phase.raise_objection(this); ... phase.drop_objection(this);
`uvm_info("TAG","msg",UVM_MEDIUM)  `uvm_error  `uvm_fatal  `uvm_warning
```

Q	A
Why the factory?	Swap a class for a derived one **without editing the testbench** — override by type or by instance path from the test.
Why objections?	They keep `run_phase` alive; simulation ends when the last objection drops.
Why `config_db`?	Passes handles/settings down the hierarchy without wiring constructors together.
Why a **virtual** interface?	Classes are dynamic and have no static hierarchy — a virtual interface is the handle that lets class-based TB code touch real DUT pins.
Why `program` blocks / clocking blocks?	They put TB activity in the Reactive region, eliminating driver/DUT races on the same edge.

9.7 Verification Concepts Glossary

Term	Meaning
DV plan / test plan	features → tests → coverage bins, tracked to closure
Golden / reference model	an independent (often C/SystemC) implementation for the scoreboard
Constrained random	random within legal constraints; finds bugs directed tests miss
Coverage-driven closure	run randoms → look at holes → add constraints/directed tests
Formal / model checking	proves a property for **all** inputs; great for arbiters, FIFOs, CDC, and for finding deep corner cases simulation cannot reach
Emulation / FPGA prototyping	runs real software at MHz speeds, for system-level bugs
x-propagation	2-state sim hides x; use x-prop mode to catch uninitialised-reset bugs that RTL sim optimistically resolves
Gate-level sim (GLS)	catches x-issues, reset problems, and timing with SDF back-annotation
Regression	nightly run of all tests with many random seeds
Bug curve	bug-discovery rate flattening is a tape-out readiness signal

// §10 RAPID-FIRE INTERVIEW Q&A

10.1 Language & Simulation

Question	Answer
`$display` **vs** `$strobe` **vs** `$monitor` E	`$display` prints immediately in the Active region (so it can show pre-NBA values); `$strobe` prints once at the end of the timestep with final values; `$monitor` auto-prints at the end of every timestep in which any of its arguments changed — only one can be active.
`==` **vs** `===` E	`==` returns `x` if either operand contains `x`/`z`; `===` compares `x` and `z` literally and always returns 0 or 1. `===` is not synthesisable — no gate can "see" an `x`.
Blocking vs non-blocking E	`=` evaluates and updates immediately and blocks the next statement → combinational. `<=` samples the RHS now but defers the update to the NBA region → all flops in the design see the **old** values, which is exactly how real registers behave.
Why does `<=` **prevent races?** M	All RHS values are sampled before any LHS is updated, so the order in which the simulator runs the `always` blocks cannot change the result.
Event queue regions M	Active → Inactive (`#0`) → NBA → Postponed (`$strobe`/`$monitor`). SV adds Preponed (assertion/clocking sampling), Observed (assertion evaluation) and Reactive (program blocks).
`case` **vs** `casez` **vs** `casex` M	Exact 4-state match / `?` and `z` are don't-care / `x` **and** `z` are don't-care. `casex` is banned: a real `x` in the selector silently matches the first don't-care branch and masks the bug.
Reduction vs bitwise E	Bitwise are binary and produce a vector of the same width; reduction are unary and collapse one vector to 1 bit. `a&b` = 4 bits; `&a` = 1 bit.

Question	Answer
Function vs task E	Function: zero time, one return value, callable in an expression, synthesisable. Task: may consume time, any number of outputs, called as a statement, usually TB-only.
`automatic` **vs** `static` H	Static (Verilog default) shares one copy of locals — concurrent calls corrupt each other and recursion is impossible. `automatic` allocates per call on the stack → re-entrant and recursive.
What does `generate` **do?** E	Replicates or conditionally includes **hardware** at elaboration time using `genvar`/parameters. It is not a runtime loop.
`parameter` **vs** `localparam` **vs** `` `define `` E	Overridable per instance / derived and not overridable / global textual macro with no scope or type.
Signed arithmetic pitfall M	An expression is signed only if **every** operand is signed. One unsigned operand makes the whole expression unsigned → zero-extension instead of sign-extension. Fix with `$signed()`.
`#5 a=b;` **vs** `a=#5 b;` M	Inter-assignment: wait 5, **then** sample `b`. Intra-assignment: sample `b` now, wait 5, then assign. The intra form models transport delay.
Multiple drivers on a `wire`**?** E	Resolved by the net's strength/value table: 0+1 → `x`; `z`+value → value; `wand`/`wor` give wired-AND/OR; stronger strength always wins.
What is `` `timescale 1ns/1ps`**?** E	Time unit / precision for delays in that file. The finest precision anywhere in the design sets the simulator tick and dominates runtime.

10.2 Synthesis & RTL

Question	Answer
Latch vs flip-flop E	Latch is transparent while its enable is active and is inferred by an incomplete `if`/`case`; a flop samples only on an edge. Latches are hard to time (transparency/borrowing), pass glitches and hurt DFT.
How do I avoid a latch? E	Assign every output on every path: a default assignment at the top of the block, a `default` in every `case`, an `else` on every `if`. Use `always_comb` so the tool warns you.
Does `reg` **always become a register?** E	No. `reg` is only a variable. A register is created only by an edge-sensitive block (or by an incomplete comb block, which makes a latch).
Moore vs Mealy E	Moore output = f(state): registered, glitch-free, one cycle later, more states. Mealy output = f(state, input): same-cycle response, fewer states, but a combinational path from input to output.
One-hot vs binary encoding M	One-hot: N flops, single-level decode, best f_{max}, ideal on FPGA. Binary: $\lceil \log_2 N \rceil$ flops, deeper decode, best area. Gray: minimum switching, mandatory for async pointers.
Why avoid `full_case/parallel_case`**?** H	They assert facts the RTL doesn't enforce, so simulation and gates can diverge. Use a real `default` plus `unique/priority case`, which check the same assumptions at run time.
Priority vs parallel logic M	`if/else if` chains build a priority (cascaded mux) structure; a `case` with mutually exclusive labels builds a flat parallel mux. Priority chains are longer paths.
Sync vs async reset M	Sync: needs a clock, filters glitches, adds a gate to the D path. Async: works with no clock, no data-path cost, but glitch-sensitive and needs recovery/removal. Best practice: async assert, sync de-assert.
Recovery and removal? M	Recovery = minimum time reset must be de-asserted **before** the clock edge. Removal = minimum time it must stay asserted **after** the edge. They are setup/hold for reset release.
Why register module outputs? M	It makes each block's timing budget self-contained, so blocks can be timed and optimised independently and integration doesn't create surprise cross-block critical paths.
Combinational loop — why bad? E	It has no defined stable value: it oscillates or latches unpredictably, cannot be timed by STA, and hangs simulation in a zero-delay loop.
Can I use a `for` **loop in RTL?** E	Yes, if the bounds are compile-time constants — it is unrolled into parallel hardware. It is not a sequential loop.

10.3 Timing, CDC & Physical

Question	Answer
Setup and hold E	Setup: data must be stable *before* the edge → `t_cq + t_comb + t_su ≤ T + skew`. Hold: data must be stable *after* the edge → `t_cq,min + t_comb,min ≥ t_hold + skew`.
Fix a hold violation by slowing the clock? M	**No** — the period cancels out of the hold equation. Add delay (buffers) to the data path or reduce skew. Setup violations, by contrast, always yield to a slower clock.
Skew: good or bad? M	Positive skew (capture clock arrives late) helps setup and hurts hold; negative skew does the reverse. "Useful skew" deliberately exploits this to borrow time between stages.
What is metastability? M	Sampling a signal that violates setup/hold leaves the flop at an intermediate voltage for an unbounded settling time; downstream logic can read it differently, corrupting state.
Why exactly 2 flops? M	MTBF grows **exponentially** with settling time, and each extra stage buys a whole clock period. Two stages is plenty at ordinary frequencies; add a third above ~500 MHz or for safety-critical paths.
Can I put logic between sync stages? M	No — it consumes settling time and collapses the MTBF. Synchroniser flops must be adjacent and are usually marked (`ASYNC_REG`) so P&R keeps them close.
Why can't I 2-FF a multi-bit bus? H	Each bit resolves independently, so the destination can capture a value that never existed (skew across bits). Use Gray code, an async FIFO, or a handshake with the data held stable.
Fast→slow pulse crossing? H	A pulse shorter than the destination period is missed. Convert it to a level with a toggle flop, synchronise the level, then edge-detect on the far side.
Why Gray pointers in an async FIFO? H	Only one bit changes per increment, so a mid-transition sample yields either the old or the new pointer — both safe. The flags are then pessimistic, never optimistic.
Why N+1 pointer bits? H	The extra MSB distinguishes full from empty when the lower bits are equal.
Glitch-free clock gating? M	Latch the enable on the **low** phase of the clock, then AND with the clock. A bare `clk & en` makes runt pulses if `en` moves while the clock is high.
Setup/hold on a clock-gate?	Yes — the enable must meet setup to the gating latch, which is why ICG cells are characterised and instantiated rather than hand-built.
What is a false path? M	A structurally-present path that can never be functionally exercised, so STA is told to ignore it (e.g. across a synchroniser, or into test-only logic).
Multicycle path caveat H	Relaxing setup by N cycles without also setting the hold multicycle creates hold violations — always set both.
Antenna / IR drop / EM?	Physical-design effects: charge damage during fabrication, supply voltage droop increasing delay, and electromigration wearing out wires — all fixed at P&R, not in RTL.

10.4 SystemVerilog & Verification

Question	Answer
`logic` **vs** `reg`/`wire` E	`logic` decouples the type from the assignment style: drivable from a continuous assign **or** a procedural block, with a compile-time single-driver check. Use `wire` only where you genuinely need multiple drivers.
`always_comb` **vs** `always @*` M	Runs once at time 0; is sensitive to signals read inside called functions; enforces one driver; warns on inferred latches.
`bit` **vs** `logic` E	2-state vs 4-state. `bit` is faster and uses less memory but hides `x`-propagation bugs — keep DUT-facing signals 4-state.
Packed vs unpacked M	Packed = contiguous bits, sliceable, synthesisable as a bus. Unpacked = array of elements, models memory, cannot be sliced as a whole.
`|->` **vs** `|=>` M	Overlapping: consequent checked in the same cycle the antecedent matched. Non-overlapping: checked one cycle later (identical to `|-> ##1`).
What is a vacuous pass? H	The antecedent never occurred, so the implication passed trivially. Add a `cover property` on the antecedent to prove the check ran.
Immediate vs concurrent assertion M	Immediate is an `if` inside procedural code with no clock; concurrent is clocked, samples in Preponed and evaluates in Observed.
`join` **vs** `join_any` **vs** `join_none` M	Wait for all / for the first / for none (children run when the parent next blocks).
Mailbox vs semaphore vs event E	Typed FIFO between threads / counting key for shared resources / a synchronisation trigger (use `wait(e.triggered)` to avoid missing it).
Shallow vs deep copy M	`new obj` copies handles, so both objects share the referenced sub-objects. A deep copy requires a hand-written `copy()`.
`rand` **vs** `randc` E	`randc` cycles through every value in the range before repeating any; `rand` is memoryless.
`solve...before` H	Biases the **distribution** of solutions only — it never adds or removes legal solutions.
Code vs functional coverage M	Code coverage is automatic and asks "did this line run?"; functional coverage is written from the spec and asks "did I test this feature?" Unwritten features show 100% code coverage.
Why a virtual interface? M	Class objects are dynamic and have no place in the static module hierarchy; a virtual interface is the handle through which class-based stimulus reaches real DUT pins.
Why the UVM factory? M	It lets a test substitute a derived class (e.g. an error-injecting driver) for a base class without modifying the environment.
Why objections? E	They hold `run_phase` open; the simulation ends when the last one is dropped.
Formal vs simulation H	Formal proves a property exhaustively for all inputs (or gives a counterexample) but struggles with large state spaces; simulation scales but only covers the stimulus you generate. Use formal on arbiters, FIFOs, CDC and protocol logic.

// §11 "READ THE CODE" PUZZLES

P1. What prints at t=20?

E

```
reg [3:0] a, b;
initial begin
   a = 4'd2;  b = 4'd5;
   #10;  a <= b;  b <= a;
   #10;  $display("a=%0d b=%0d", a, b);
end
```

A **a=5 b=2.** Both RHS are sampled at t=10 using the *old* values, then both LHS update in the NBA region — a clean swap. With `=` instead you would get `a=5 b=5`.

P2. What hardware is this?

M

```
always @(posedge clk) begin
   a = in_val;   b = a;   out <= b;
end
```

A **One flip-flop,** `out` ← `in_val`. The blocking assignments complete instantly, so `b` already equals `in_val` when the NBA is scheduled. `a` and `b` are optimised away (unless read elsewhere).

P3. Why is this NOT a 2-stage shift register?

M

```
always @(posedge clk) begin
   q1 = d;    q2 = q1;
end
```

A **Both** `q1` **and** `q2` **sample** `d` **— one stage of delay, not two.** Synthesis goes further and merges the two identical registers into a *single* flop driving both outputs (Vivado: one `FDRE`, `D` ← `d`, `Q` → `q1`, `q2`). Blocking assignment makes `q1` visible immediately. Swap to `<=` for a real 2-stage shifter. (Amusingly, writing the two lines in *reverse* order with `=` **does** give a shift register — which is precisely why the rule is "never use `=` in a clocked block".)

P4. What is inferred, and what is the bug?

E

```
always @(a or b or sel)
   if (sel) out = a + b;
```

A **Transparent latch on** `out`. No assignment when `sel==0`, so the old value must be remembered. Fix with `else out = '0;` or a default `out = '0;` before the `if`.

P5. Final value of `arr`?

H

```
integer i;   reg [3:0] arr [0:3];
initial begin
   for (i = 0; i < 4; i = i + 1)
      fork  arr[i] = i;  join_none
   #10;  $display("%0d %0d", arr[0], arr[3]);
end
```

A **All four entries stay** `x`. `join_none` threads do not run until the parent blocks — that is at `#10`, by which time the loop has finished and `i==4`. All four threads write `arr[4]`, which is out of bounds and silently discarded. Fix: `for (int i=0; ...)` so each iteration owns its copy, or copy `i` into an `automatic` local inside the fork.

P6. Value of `cnt` after 3 clocks?

M

```
reg [2:0] cnt = 3'b000;
always @(posedge clk) begin
   cnt <= cnt + 1'b1;
   cnt <= cnt + 2'b10;
end
```

A **cnt = 6** (`3'b110`). Both NBAs sample the same old `cnt`, and the **last** update scheduled to a given variable wins. So the effective behaviour is `cnt <= cnt + 2`: `0` → `2` → `4` → `6`.

P7. Predict both lines.

M

```
reg [7:0] a = 8'hA5;                  // 1010_0101
$display("%h", a >> 2);
$display("%h", {{2{a[7]}}, a[7:2]});
```

A **29** then **e9**. `>>` is a logical shift that fills with zeros → `0010_1001`. The concatenation replicates the sign bit twice → `1110_1001`, i.e. an arithmetic shift. (`a >>> 2` only sign-extends if `a` is declared `signed`.)

P8. What do these print?

E

```
reg [3:0] a = 4'b1010, b = 4'b0101;
$display("%b", a && b);   $display("%b", a & b);
reg [3:0] v = 4'b1101;
$display("%b %b %b", ^v, ~|v, ~v);
```

A `1`, then `0000`; then `1 0 0010`. `&&` asks "both non-zero"; `&` is bitwise. `^v` = parity of `1101` = `1` (three ones, odd); `~|v` = "v is zero" = `0`; `~v` = `0010`.

P9. Which counter is this?

E

```
always @(posedge clk) begin
   cnt <= cnt + 1;
   if (cnt == 4'd9) cnt <= 4'd0;
end
```

A **Modulo-10 (BCD) counter.** Statements in one block have procedural priority: the second NBA overrides the first when `cnt==9`. This is the same "last write wins" rule as P6 — here used deliberately.

P10. What memory behaviour is this?

M

```
always @(posedge clk) begin
   if (we) mem[addr] <= din;
   dout <= mem[addr];
end
```

A **Synchronous single-port RAM, read-first (old data).** Both are NBAs sampling the pre-edge contents, so on a simultaneous read+write `dout` returns the value that was there *before* the write. For write-first, assign `dout <= din;` inside the `if`.

P11. Find the race.

H

```
initial begin clk = 0;  forever #5 clk = ~clk;  end
initial begin data = 8'h00;  @(posedge clk);  data = 8'hFF;  end
```

A **Testbench/DUT sampling race.** `data = 8'hFF` lands in the Active region of the very edge the DUT's flops sample, so the DUT may capture `00` or `FF` depending on simulator ordering. Fix: drive with `data <= 8'hFF;`, or use a clocking block with an output skew:

```
default clocking cb @(posedge clk);
   default input #1step output #2ns;
   output data;
endclocking
```

P12. Pass or fail?

E

```
// t=50 (posedge): req=1, gnt=0
// t=60 (posedge): req=0, gnt=1
assert property (@(posedge clk) req |-> ##1 gnt);
```

A **Pass.** The antecedent `req` holds at t=50, so the consequent is checked exactly one cycle later at t=60, where `gnt` is high. At t=60 `req` is low, so no new attempt starts.

P13. What does this generate block build?

M

```
genvar i;
generate for (i = 0; i < 4; i = i + 1) begin: bit_slice
   if (i == 0) assign out[i] = in[i];
   else        assign out[i] = in[i] ^ out[i-1];
end endgenerate
```

A **Gray-to-binary converter** — a cumulative XOR (prefix) chain: `out[0]=in[0]`, `out[1]=in[1]^out[0]`, `out[2]=in[2]^out[1]`, `out[3]=in[3]^out[2]`. Note it is a 4-deep XOR chain, not a balanced tree, so it is the slow form. The *binary-to-Gray* direction is the cheap one: `g = b ^ (b>>1)`, one XOR deep.

P14. Is this reset synchroniser correct?

M

```
always @(posedge clk or posedge rst) begin
   if (rst) begin q1 <= 1'b0;  q2 <= 1'b0;  end
   else     begin q1 <= 1'b1;  q2 <= q1;    end
end
```

A **Functionally yes — this is the textbook reset bridge** (async assert, synchronous de-assert): `q2` is the clean reset, released two clocks after `rst` drops. Two real-world caveats: ① the flops must carry an `ASYNC_REG`-style attribute so place & route keeps them adjacent and does not duplicate them, otherwise MTBF degrades; ② `rst` itself must be glitch-free — if it comes from combinational logic, it will inject spurious resets.

P15. Why does this hang the simulator?

M

```
always @(*)  out = out + in;
```

A **Zero-delay combinational feedback.** `out` is both read and written, so every update re-triggers the block within the same timestep — an infinite Active-region loop. In hardware it is a combinational loop, which STA cannot analyse at all.

P16. Can a glitch on `comb_in` reach `out_pulse`?

M

```
always @(posedge clk) begin
   q1 <= comb_in;
   out_pulse <= comb_in & ~q1;
end
```

A **No.** `out_pulse` is a registered output, so it only changes on a clock edge; a glitch that settles before the setup window is invisible. **But** a glitch landing *inside* the setup/hold aperture can drive that flop metastable — which is exactly why an asynchronous `comb_in` must be synchronised first.

P17. What is printed?

H

```
reg [3:0] m = 4'd15, n = 4'd15;   reg [7:0] p;
p = m * n;                        $display("%0d", p);
                                  $display("%0d", m * n);
```

A **225**, then **1**. In the assignment the LHS is 8 bits, so the multiply is evaluated at 8 bits → 225. Inside `$display` the expression is **self-determined**: it is evaluated at 4 bits, so 225 mod 16 = 1. Always widen explicitly: `$display("%0d", 8'(m)*n);`

P18. Which of these four is a real 3-stage pipeline?

M

```
// A  always_ff @(posedge clk) begin c<=b; b<=a; a<=d; end
// B  always_ff @(posedge clk) begin a<=d; b<=a; c<=b; end
// C  always_ff @(posedge clk) begin a=d;  b=a;  c=b;  end
// D  always_ff @(posedge clk) begin c=b;  b=a;  a=d;  end
```

A **Three stages from A, B and D; only one from C.** With NBAs (A, B) statement order is irrelevant — that is the whole point. With blocking, order decides: D happens to write in reverse dependency order so it still builds three flops, while C collapses to a single one. Only A and B are acceptable RTL.

// §12 CLASSIC GOTCHAS & TRAPS

#	Trap	What actually happens / cure
1	Incomplete sensitivity list `always @(a or b)` with c used inside	Simulation ignores c; synthesis builds a 3-input function → sim/synth mismatch. Use `always_comb`.
2	Two clocks in one block `@(posedge clkA or posedge clkB)`	No standard cell has two clock pins — unsynthesisable unless one edge is used as an async reset/preset.
3	Blocking assignment shared across two `always` blocks	Result depends on the simulator's arbitrary block ordering. Use `<=`.
4	Combinational self-feedback `out = out + in`	Infinite zero-delay loop / combinational loop in gates.
5	`case` without `default`	Latch on every variable not assigned in some branch.
6	Silent width truncation `sum = a + b` with all 8-bit	Carry-out lost. Widen first: `{1'b0,a} + {1'b0,b}` into a 9-bit result.
7	Mixing signed and unsigned	The whole expression goes unsigned → zero-extension. Wrap with `$signed()`.
8	Both `assign` and `always` driving one signal	Illegal in Verilog (`reg` can't be `assigned`); compile error on a SV `logic`.
9	Combinationally-generated async reset	A glitch on the OR gate resets the chip. Pass it through a reset synchroniser first.
10	Mixing `=` and `<=` on one variable	Undefined-ish scheduling, unpredictable synthesis. Never do it.
11	Missing `begin/end` after `if`	Only the first statement is conditional — a silent functional bug that looks correct when indented.
12	Implicit wire from a typo'd port name	1-bit net, silently mis-connected. Use `` `default_nettype none ``.
13	`x` optimism in RTL sim	`if (x)` takes the `else` branch, so RTL "works" while gates are undefined. Use x-prop mode + gate-level sim.
14	Non-reset state register	FSM starts at `x`, and `x`-optimism hides it until GLS. Always reset state, valid and control bits.
15	Reading an unpacked array in `always_comb` sensitivity	Some tools only trigger on the indexed element; be explicit.
16	`for` loop with a non-constant bound in RTL	Not synthesisable — the tool cannot unroll it.
17	`#0` used to "order" events	Just moves the race into the Inactive region. Restructure instead.
18	Concatenating an unsized literal `{a, 1}`	`1` is 32 bits wide inside a concatenation. Always size literals: `{a, 1'b1}`.

// §13 COMPANY-STYLE PROBLEMS

C1 · NVIDIA — parameterised fixed-priority arbiter

M

Given `req[N-1:0]`, produce a one-hot `gnt[N-1:0]` with bit 0 highest priority. Combinational, scalable, no long loop.

```
module prio_arb #(parameter N = 4)
  (input logic [N-1:0] req, output logic [N-1:0] gnt);
  // isolate lowest set bit -> exactly one-hot, or all-zero
  assign gnt = req & (~req + 1'b1);
endmodule
```

A **Talk track:** `req & -req` is elegant but still synthesises to an N-deep borrow chain, so for large N say you would build a **parallel-prefix OR** ("has any lower request?") tree, giving O(log N) depth: `gnt[i] = req[i] & ~any_lower[i]` where `any_lower` is a Kogge-Stone style prefix. Then mention **round-robin** (mask + rotate, see §4.3) to answer the inevitable follow-up about starvation.

C2 · Texas Instruments — slow, glitchy asynchronous sensor input

M

A 100 kHz external signal with slow edges and occasional glitches enters a 200 MHz core. Why isn't a 2-FF synchroniser enough, and what must precede it?

A ① **A slow edge** lingers in the CMOS threshold band for many core cycles: the input buffer oscillates, burns crowbar current, and different flops can capture different values on the same edge. ② **Glitches** narrower than the setup+hold aperture violate timing outright. So put an analog **Schmitt trigger** (hysteresis) on the pad to sharpen the edge, then a **digital glitch filter / debouncer** (require the level to persist for K consecutive samples), and only then the 2-FF synchroniser. The synchroniser fixes metastability; it does **not** fix slew or bounce.

C3 · AMD — which circuit is a glitch-free clock gate?

M

A `assign gclk = clk & en;` · **B** flop `en` on `posedge clk`, then AND with `clk` · **C** negative-level latch on `en` (transparent while `clk==0`), then AND with `clk` · **D** OR of `clk` with an inverted enable register

A **Correct answer: C** — the standard ICG. While `clk` is low the latch is transparent and absorbs the new `en`; when `clk` rises the latch closes, holding `en` stable for the entire high phase, so the AND output can never produce a runt pulse. **A** glitches whenever `en` moves while `clk` is high. **B** changes `en` at the same instant the clock rises — a race that can chop the first pulse. **D** has the same hazard on the opposite phase. In practice you instantiate the library ICG rather than writing it.

C4 · ARM — pipeline stall and bubble insertion

H

```
always_ff @(posedge clk or negedge rst_n)
  if (!rst_n) begin  s1 <= '0;  s2 <= '0;  end
  else if (!hazard) begin
      s1 <= s1_in;              // stage 1 advances
      s2 <= s1;                 // stage 2 advances
  end else begin
      s1 <= s1;                 // STALL  : hold stage 1
      s2 <= '0;                 // BUBBLE : inject a NOP into stage 2
  end
```

A **Talk track:** a stall must **freeze everything upstream** of the hazard and **inject a bubble** downstream, otherwise you either lose an instruction or duplicate one. Expect the follow-ups: *what else could you do instead of stalling?* → **forwarding/bypassing** (route the ALU result straight back, removing most RAW stalls) and, for loads, a single unavoidable **load-use** stall. Also mention that `s1 <= s1;` synthesises to a clock-enable, not a mux, on most libraries.

C5 · Samsung — multiplier bit growth

M

6-bit **signed** A × 8-bit **unsigned** B. Minimum output width?

A Operand A ∈ [-32, +31] and B ∈ [0, 255] → product ∈ [-8160, +7905]. A 14-bit two's-complement word spans [-8192, +8191], so **14 bits is the mathematically tight answer**. But the **synthesis-safe rule** is: to multiply signed × unsigned in Verilog you must first zero-extend B by one bit to make it a 9-bit *signed* value, and signed M × signed N needs M+N — giving **15 bits**. Say both: "14 is tight, 15 is what the standard M+N+1 rule gives you and what I would code, because it never needs a range proof."

```
wire signed [14:0] p = $signed(a) * $signed({1'b0, b});
```

C6 · Cadence — write the protocol assertions

H

"After `frame_valid` rises, `data_valid` must assert within 1–3 cycles. Once `data_valid` is high, `frame_valid` must stay high until `transfer_done`."

```
property p_latency;
   @(posedge clk) disable iff (!rst_n)
   $rose(frame_valid) |-> ##[1:3] data_valid;
endproperty
a_latency : assert property (p_latency);
c_latency : cover  property (p_latency);   // guard vs vacuity

property p_hold;
   @(posedge clk) disable iff (!rst_n)
   (frame_valid && data_valid) |->
        frame_valid until_with transfer_done;
endproperty
a_hold : assert property (p_hold);
```

A **Why** `until_with`: `frame_valid` must still be high *on* the cycle `transfer_done` asserts; plain `until` would exclude that final cycle. Add the `cover` so a vacuous pass can't hide a testbench that never raised `frame_valid`.

C7 · Intel/Qualcomm — divide-by-3 with 50% duty cycle

M

```
always_ff @(posedge clk) p <= (cnt < 1);   // cnt: 0,1,2 rollover
always_ff @(negedge clk) n <= (cnt < 1);   // same, half cycle later
assign clk_div3 = p | n;                   // 50% duty
```

A An odd division cannot be 50% from posedge logic alone, because the duty boundary falls half a cycle in. Two dividers — one on each edge — OR'd together shift by T/2 and produce an exact 50% output. **Then say the real answer:** in an ASIC you don't build clocks in RTL. Generate a **clock enable** pulse every 3rd cycle and keep one clock domain, or use a PLL/clock-divider macro — a hand-built divided clock is unbalanced, breaks scan and complicates STA.

C8 · Broadcom/Marvell — synchronous FIFO, then make it async

H

```
// Synchronous FIFO, depth 2**AW, one clock
logic [AW:0] wptr, rptr;            // extra MSB = wrap bit
always_ff @(posedge clk or negedge rst_n)
  if (!rst_n) begin wptr <= '0;  rptr <= '0;  end
  else begin
     if (push && !full) begin
        mem[wptr[AW-1:0]] <= din;   wptr <= wptr + 1;   end
     if (pop  && !empty)           rptr <= rptr + 1;
  end
assign dout  = mem[rptr[AW-1:0]];
assign empty = (wptr == rptr);
assign full  = (wptr[AW] != rptr[AW]) &&
               (wptr[AW-1:0] == rptr[AW-1:0]);
assign count = wptr - rptr;                 // works across wrap
```

A **Making it asynchronous** (see §6.4): split into two clock domains; convert each pointer to **Gray** before it crosses; pass each through a 2-FF synchroniser; compute `full` in the write domain and `empty` in the read domain using the synchronised remote pointer. The flags become **pessimistic but never optimistic**. Memory must be a dual-port RAM with independent clocks and **no** reset on the array.

// §14 VERILOG VS SYSTEMVERILOG VS VHDL

Criterion	Verilog-2001	SystemVerilog	VHDL
Typing	weak; implicit resize & truncation	weak data, strong user types/structs/enums	very strong; explicit conversions everywhere
Verbosity	terse, C-like	terse, C-like + OOP	verbose, Ada-like
Verification	basic testbenches	OOP, constrained random, coverage, SVA, UVM	needs OSVVM / UVVM / VUnit
Intent checking	`always` for everything	`always_comb/ff/latch`	explicit `process` with sensitivity
Where used	legacy IP maintenance	mainstream ASIC/FPGA design & verification	aerospace, defence, European & FPGA houses

// §15 FPGA VS ASIC — WHAT CHANGES IN YOUR RTL

Topic	FPGA	ASIC
Flip-flops	abundant and free — favour **one-hot** FSMs and heavy pipelining	each flop costs area and clock-tree power — favour binary encoding
Reset	FFs power up to a known state from the bitstream; prefer **synchronous** reset, and reset only what you must	state is undefined at power-up; you **must** reset all control logic
`initial` blocks	synthesisable for RAM/ROM/register initial values	ignored — never rely on them
Memory	infer block RAM by matching the vendor's template exactly, else you get a huge LUT-based array	instantiate compiled SRAM macros
Clocking	MMCM/PLL + global clock buffers (BUFG); limited number of clock domains	PLLs, custom clock tree synthesis
Clock gating	use `BUFGCE` or just a clock enable — fine-grained gating is often counter-productive	ICG cells everywhere; a major power lever
Multipliers	hard DSP slices — match their pipeline depth to get full speed	synthesised or custom
Tri-state	internal tri-states do not exist — the tool converts them to muxes; only I/O pads are real	rare internally; pads only
Iteration cost	minutes; re-programmable	months and millions — verification is everything

// §16 LOW-POWER DESIGN

$$P_{total} = \alpha \cdot C \cdot V_{DD}^2 \cdot f \;+\; V_{DD} \cdot I_{short} \;+\; V_{DD} \cdot I_{leak}$$

Technique	Attacks	Cost / caveat
Clock gating (ICG)	dynamic — the clock tree is often 30–40% of total power	enable must meet setup to the gate; needs an ICG cell
Operand isolation	dynamic — stops datapath toggling when the result is unused	extra AND gates on the critical path
Power gating (sleep transistors)	leakage — turns the block off completely	needs isolation cells, retention flops, a power-up sequence
Multi-V_t	leakage — high-V_t cells on non-critical paths	high-V_t is slower; tool trade-off
DVFS	dynamic, quadratically in V_{DD}	needs level shifters, regulators, software control
Gray / one-hot bus encoding	dynamic — fewer toggles per transition	more decode logic
Reduce logic depth / glitches	dynamic — glitches are wasted transitions	balanced trees and pipelining

UPF/CPF describe the power intent (domains, isolation, retention, level shifters) outside the RTL, so the same RTL is used for functional and low-power flows. **Isolation cells** clamp the outputs of a powered-down block; **retention flops** keep state on a small always-on supply; **level shifters** bridge two voltage domains.

// §17 DESIGN FOR TEST (DFT)

Term	What it is
Scan chain	every flop is replaced by a scan flop with a mux on D; in test mode they form a giant shift register, so any state can be loaded and observed
ATPG	automatic test-pattern generation, usually for the stuck-at and transition fault models
Coverage	fault coverage is typically required to be > 98–99% for production test
MBIST	on-chip memory self-test (March algorithms) — RAM cannot be scan-tested
LBIST	logic BIST using an on-chip LFSR pattern generator and a MISR signature compactor
Boundary scan	JTAG (IEEE 1149.1) — tests board-level interconnect through TDI/TDO/TMS/TCK
Test compression	decompressor/compactor around the scan chains to cut tester time and pin count

RTL rules that keep DFT happy: no latches (poor controllability/observability) · no internally generated or gated clocks without a test bypass (`test_mode` must force the ICG on) · no asynchronous reset driven by internal logic without a bypass · no combinational feedback loops · avoid multi-cycle and false paths where you can. Every clock and reset must be controllable from a primary input during scan shift.

// §18 BUS PROTOCOLS & A SELF-CHECKING TESTBENCH

Bus	Character	Key signals
APB	simple, non-pipelined, 2+ cycles per transfer; for slow peripherals	`PSEL PENABLE PWRITE PADDR PWDATA PREADY PSLVERR`
AHB	pipelined address/data phase, single outstanding, burst support	`HTRANS HBURST HADDR HWDATA HREADY HRESP`
AXI4	five independent channels, out-of-order via IDs, bursts up to 256 beats	`AW AR W R B` channels, each with `VALID/READY`
AXI-Stream	point-to-point data only, no addresses	`TVALID TREADY TDATA TLAST TKEEP`

The handshake rule that gets asked every time: a transfer occurs on a rising clock edge when `VALID` and `READY` are both high. **VALID must never wait for READY** (that deadlocks two connected masters), **READY may wait for VALID**, and once `VALID` is asserted it must stay asserted with stable payload until the transfer completes.

```
`timescale 1ns/1ps
module tb;
  localparam CLK = 10;                        // 100 MHz
  logic clk = 0, rst_n = 0;
  logic [7:0] din, dout;  logic vld, rdy;
  int errors = 0;

  always #(CLK/2) clk = ~clk;                 // free-running clock

  dut u_dut (.clk(clk), .rst_n(rst_n), .din(din),
             .vld(vld), .rdy(rdy), .dout(dout));

  // ---- reset: assert async, release away from an edge ----
  initial begin
     rst_n = 1'b0;  repeat (5) @(posedge clk);
     rst_n <= 1'b1;
  end

  // ---- stimulus : drive with <= to avoid the sampling race ----
  task automatic send (input [7:0] d);
     begin
        @(posedge clk);  din <= d;  vld <= 1'b1;
        do @(posedge clk); while (!rdy);   // wait for handshake
        vld <= 1'b0;
     end
  endtask

  // ---- reference model + scoreboard ----
  logic [7:0] expect_q [$];
  always @(posedge clk) if (vld && rdy) expect_q.push_back(model(din));
  always @(posedge clk) if (dout_valid) begin
     automatic logic [7:0] exp = expect_q.pop_front();
     if (dout !== exp) begin
        errors++;
        $error("t=%0t got %h exp %h", $time, dout, exp);
     end
  end

  // ---- protocol assertion ----
  assert property (@(posedge clk) disable iff (!rst_n)
     (vld && !rdy) |=> (vld && $stable(din)) );

  initial begin
     $dumpfile("tb.vcd");  $dumpvars(0, tb);
     wait (rst_n);
     repeat (100) send($urandom_range(0,255));
     repeat (20) @(posedge clk);
     if (errors) $fatal(1, "FAILED with %0d errors", errors);
     else        $display("PASSED");
     $finish;
  end
endmodule
```

// §19 LAST-MINUTE REVISION PANEL

The ten things you will almost certainly be asked: ① blocking vs non-blocking (and *why*); ② how a latch gets inferred and how to prevent it; ③ Moore vs Mealy and write an FSM; ④ setup/hold, and "can slowing the clock fix hold?"; ⑤ metastability, the 2-FF synchroniser and MTBF; ⑥ CDC for 1-bit vs multi-bit; ⑦ async FIFO with Gray pointers, full/empty derivation; ⑧ sync vs async reset and the reset synchroniser; ⑨ `casex` vs `casez` and the `x`-masking problem; ⑩ bit-width growth and signed/unsigned mixing.

Quantity	Rule
Unsigned N + N	N + 1 bits
Sum of K N-bit values	N + $\lceil \log_2 K \rceil$ bits
M × N (both signed, or both unsigned)	M + N bits
signed M × unsigned N	M + N + 1 bits (zero-extend the unsigned one first)
N-bit counter period	2^N; max-length LFSR = $2^N - 1$
2^N-deep async FIFO pointer	N + 1 bits, Gray coded
States needing E flops	binary $\lceil \log_2 N \rceil$ · one-hot N · Johnson N/2
Async FIFO latency	≈ 2–3 destination clocks through the synchroniser
Synchroniser stages	2 normally, 3 above ~500 MHz or safety-critical
Reset hold time	≥ 2–3 clocks in the **slowest** domain

How to answer a "design this block" question: ① restate the spec and ask about the interface, reset style, clock domains and throughput; ② draw the block diagram and name the state elements; ③ write the RTL with the 3-block FSM / `always_ff` + `always_comb` discipline; ④ state the reset strategy and what is *not* reset; ⑤ mention the critical path and one way to pipeline it; ⑥ list two assertions and two coverage points you would add. Steps ① and ⑥ are what separate candidates — almost nobody does them.

Red flags interviewers listen for: "`reg` means register" · using `=` in a clocked block · no `default` in a `case` · a 2-FF synchroniser on a multi-bit bus · gating a clock with an AND gate · "I'd slow the clock to fix the hold violation" · claiming `===` is synthesisable · forgetting `disable iff (!rst_n)` on an assertion.

// §20 ACRONYM & JARGON DECODER

Term	Expansion — what it means in practice
RTL	Register Transfer Level — the abstraction where you describe registers and the combinational logic between them
HDL	Hardware Description Language (Verilog, SystemVerilog, VHDL)
ASIC	Application-Specific Integrated Circuit — a fixed, fabricated chip
FPGA	Field-Programmable Gate Array — reconfigurable LUT/FF fabric
LUT	Look-Up Table — the FPGA primitive that implements combinational logic
FF / DFF	Flip-Flop / D-type flip-flop — the edge-triggered storage element
FSM	Finite State Machine (Moore or Mealy)
LFSR	Linear Feedback Shift Register — cheap pseudo-random / fast counter
FIFO	First-In First-Out buffer; *async* FIFO crosses two clock domains
CDC	Clock Domain Crossing — any signal moving between unrelated clocks
MTBF	Mean Time Between Failures — the metastability reliability metric
STA	Static Timing Analysis — exhaustive path timing without simulation
SDC	Synopsys Design Constraints — the clock/IO/exception constraint file
PVT	Process, Voltage, Temperature — the corners STA is signed off across
OCV / AOCV	(Advanced) On-Chip Variation — derating between launch and capture paths
P&R	Place and Route — physical implementation after synthesis
ICG	Integrated Clock Gating cell — the glitch-free latch+AND clock gate
PLL / MMCM	Phase-Locked Loop / Mixed-Mode Clock Manager — clock generation macros
CTS	Clock Tree Synthesis — balancing clock arrival across the die
DFT	Design For Test — scan, ATPG, BIST infrastructure
ATPG	Automatic Test Pattern Generation (stuck-at, transition faults)
BIST / MBIST	Built-In Self-Test / Memory BIST — on-chip test of logic or RAM
MISR	Multiple-Input Signature Register — compacts BIST responses
JTAG	IEEE 1149.1 boundary scan (TDI / TDO / TMS / TCK)
UPF / CPF	Unified / Common Power Format — power intent kept outside the RTL
DVFS	Dynamic Voltage and Frequency Scaling
GLS	Gate-Level Simulation — post-synthesis sim, often with SDF timing
SDF	Standard Delay Format — back-annotated cell and net delays
DUT	Design Under Test — the block the testbench drives
TB	Testbench
SVA	SystemVerilog Assertions — the concurrent property language
UVM	Universal Verification Methodology — the standard class library
TLM	Transaction Level Modelling — port/export communication in UVM
CRV	Constrained Random Verification
NBA	Non-Blocking Assignment (and its update region in the event queue)
RAW / WAR	Read-After-Write / Write-After-Read — pipeline data hazards
II	Initiation Interval — cycles between successive pipeline inputs
f_{max}	Maximum clock frequency the timing-critical path allows
CLA / RCA / CSA	Carry-Lookahead / Ripple-Carry / Carry-Save Adder
AXI / AHB / APB	ARM AMBA buses — streaming/high-performance, pipelined, peripheral
IP	Intellectual Property — a reusable pre-verified design block
ECO	Engineering Change Order — a late, surgical netlist fix

Two-minute pre-interview drill: say out loud, from memory — ① the blocking/non-blocking rule and *why*; ② the two STA inequalities; ③ why a hold violation ignores clock period; ④ the four CDC schemes and when each applies; ⑤ the async-FIFO full/empty derivation; ⑥ how a latch appears and the three cures. If any of the six is shaky, that is exactly where the interview will go.