

Clocked Verilog: a tutorial

How to write flip-flops, registers and counters in Verilog, and how to test them. It also covers the background Lab 6a needs: synchronizing and debouncing a push button. You already write combinational Verilog; this handout adds the clock.

How to use this handout. Each section has a short example you can copy into a simulator. Every example was compiled and simulated with Icarus Verilog. The style matches the lab handouts: a single clock `clk`, an active-high synchronous `reset`, and parameters for anything that must be small in simulation and large on the board. Exercises with answers are at the end.

1. Two kinds of `always` block
2. The D flip-flop and reset
3. Blocking and nonblocking assignment
4. Registers that hold, load and shift
5. Counters and parameters
6. Slowing down: enables, not clocks
7. Next-value logic and the latch trap
8. Remembering the past: levels and events
9. Inputs from the outside world (Lab 6a)
10. Testing clocked designs
11. Rules and common mistakes
12. Exercises

1 Two kinds of `always` block

Everything in a synchronous design is one of two things: **combinational logic**, whose output follows its inputs, or **flip-flops**, which change only at a clock edge. Verilog has one kind of `always` block for each.

	Combinational	Clocked (sequential)
Header	<code>always @(*)</code>	<code>always @(posedge clk)</code>
Runs when	any input changes	only at the rising edge of <code>clk</code>
Assignment	blocking <code>=</code>	nonblocking <code><=</code>
Builds	gates and multiplexers	flip-flops (with gates in front)
If a variable is not assigned	a latch is inferred: a bug	it holds its value: what a flip-flop does

In both cases the target is declared `reg`. The name is misleading: `reg` only means "assigned inside an `always` block". Whether it becomes a flip-flop depends on the block, not on the keyword.

2 The D flip-flop and reset

dff_sr.v

```
module dff_sr (  
    input wire clk, reset, d,  
    output reg q  
);  
    always @(posedge clk) begin  
        if (reset) q <= 1'b0;  
        else      q <= d;  
    end  
endmodule
```

`@(posedge clk)`: the block runs only at rising edges, so `q` is a positive-edge-triggered D flip-flop.

`reset` is tested *inside* the edge-triggered block, so it only acts at a clock edge: a **synchronous** reset. It is **active-high**: 1 means reset.

Between edges, `q` keeps its value whatever `d` does.

Reset style	Code	Use
Synchronous, active-high	<code>always @(posedge clk) if (reset) q <= 0;</code>	Our labs. Simple timing; reset is just another input.
Asynchronous, active-low	<code>always @(posedge clk, negedge rst_n) if (!rst_n) q <= 0;</code>	Mano's HDL examples. Acts immediately, without a clock edge.

Every register in a design should be reset to a known value. Without reset, simulation starts at `x` (unknown) and the hardware starts at whatever value power-up gives.

3 Blocking and nonblocking assignment

This is the most important rule in clocked Verilog, and the most common source of bugs.

- **Blocking** `a = b;` updates `a` immediately. The next statement already sees the new value, like a variable in C.
- **Nonblocking** `a <= b;` evaluates `b` now but updates `a` only after *every* statement triggered by this clock edge has been evaluated. All right-hand sides see the values from *before* the edge.

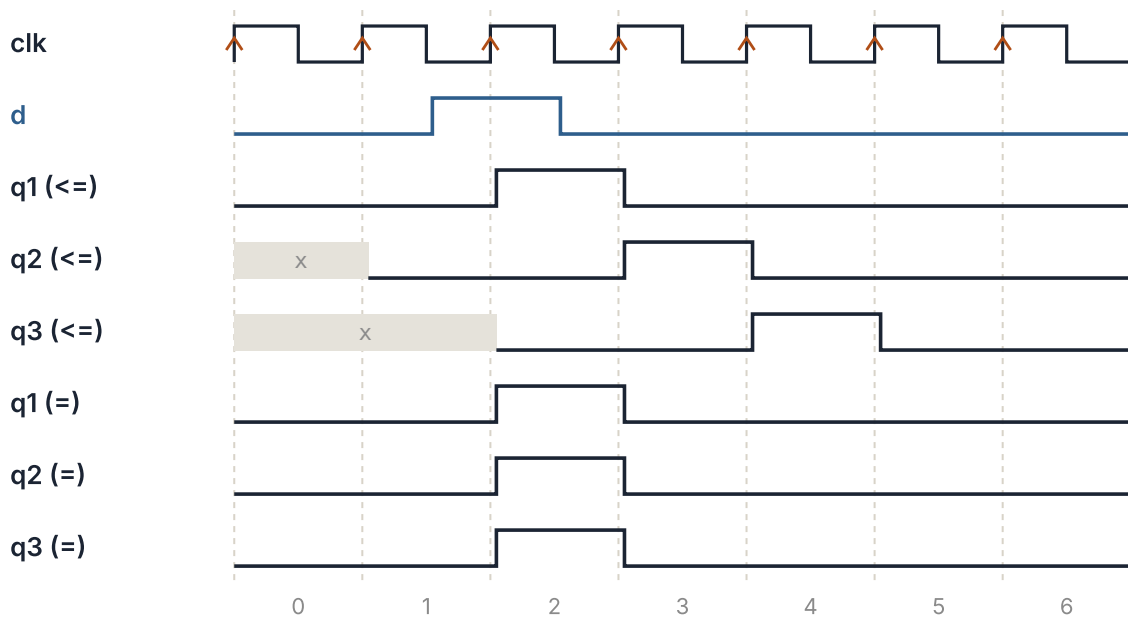
Nonblocking is exactly how real flip-flops behave: all flip-flops sample their inputs at the same instant, then all outputs change together. Compare two versions of a three-stage shift register:

Nonblocking: three flip-flops in a chain

```
module chain_nb (  
    input wire clk, d,  
    output reg q1, q2, q3  
);  
    always @(posedge clk) begin  
        q1 <= d;  
        q2 <= q1;  
        q3 <= q2;  
    end  
endmodule
```

Blocking: wrong

```
module chain_b (  
    input wire clk, d,  
    output reg q1, q2, q3  
);  
    always @(posedge clk) begin  
        q1 = d;  
        q2 = q1;  
        q3 = q2;  
    end  
endmodule
```



Simulation of both versions. A single 1 on `d` moves one stage per clock through the nonblocking chain. In the blocking version all three outputs copy `d` at the same edge: it behaves like one flip-flop. Grey = unknown (`x`) before the chain fills.

Why? In the blocking version, `q1 = d` happens first, so `q2 = q1` already sees the *new* `q1`, and so on. The synthesis tool builds what the code says: one flip-flop whose output drives `q1`, `q2` and `q3`. With nonblocking assignment the order of the three lines does not matter at all.

Swapping two registers needs no temporary

```
module swap_nb (
    input wire    clk, reset,
    output reg [3:0] a, b
);
    always @(posedge clk) begin
        if (reset) begin
            a <= 4'd3; b <= 4'd9;
        end else begin
            a <= b;    b <= a;
        end
    end
endmodule
```

Because every right-hand side is read before any register changes, `a <= b; b <= a;` swaps the two values at each edge. In simulation `a` and `b` alternate 3, 9, 3, 9, ... With `=`, both would end up equal.

The assignment rules

1. In `always @(posedge clk)`: use only `<=`.
2. In `always @(*)`: use only `=`.
3. Never mix `=` and `<=` in one block.
4. Assign each variable in exactly one `always` block. Two blocks driving the same `reg` is a multiple-driver error in synthesis.

4 Registers that hold, load and shift

An n-bit register with load

```
module reg_load #(parameter integer W = 8) (  
    input wire      clk, reset, load,  
    input wire [W-1:0] d,  
    output reg [W-1:0] q  
);  
    always @(posedge clk) begin  
        if (reset)    q <= {W{1'b0}};  
        else if (load) q <= d;  
        // no else: q holds its value  
    end  
endmodule
```

If neither `reset` nor `load` is 1, no assignment happens and `q` keeps its value. In a clocked block this "hold" is free: it is what a flip-flop does.

In hardware this is a 2-to-1 multiplexer in front of each D input, choosing between `d` and `q`: the parallel-load register from the registers lecture.

`{W{1'b0}}` is replication: W copies of a 0 bit.

A 4-bit shift register with enable

```
module shift4 (  
    input wire      clk, reset, en, sin,  
    output reg [3:0] q  
);  
    always @(posedge clk) begin  
        if (reset)    q <= 4'b0000;  
        else if (en) q <= {sin, q[3:1]};  
    end  
endmodule
```

Shifting is just concatenation. `{sin, q[3:1]}` is a new 4-bit value: `sin` at the top, then the old bits 3 to 1. Bit 0 falls off the end. A left shift would be `{q[2:0], sin}`.

An `if` chain gives priority: `reset` beats `en`. A universal shift register is the same idea with a `case` on the mode bits.

5 Counters and parameters

A mod-N counter with enable and terminal count

```
module mod_n_counter #(  
    parameter integer N = 10,  
    parameter integer W = 4  
) (  
    input wire      clk, reset, en,  
    output reg [W-1:0] count,  
    output wire      tc  
);  
    assign tc = en && (count == N-1);  
    always @(posedge clk) begin  
        if (reset)    count <= 0;  
        else if (tc) count <= 0;  
        else if (en) count <= count + 1'b1;  
    end  
endmodule
```

- `count + 1'b1` adds 1. The result is truncated to W bits, so a W-bit counter wraps from 2^W-1 to 0 on its own.
- For a modulus that is not a power of two, detect the last state, N-1, and clear. `tc` (terminal count) is 1 for exactly one cycle per N enabled cycles: it can enable a second counter or start an action.

- `tc` includes `en`, so a held counter does not keep signalling. It is a continuous `assign`: combinational, from the register's present value.
- **W must be large enough to hold N-1**: $W \geq \log_2 N$ rounded up. You can compute it with `$clog2(N)` in a `localparam`; Vivado accepts it.

Parameters: one module, two sizes

```
// on the board: 100 MHz clock, one tc every 10 ms
mod_n_counter #(.N(1_000_000), .W(20)) u_slow (.clk(clk), .reset(reset), .en(1'b1),
                                                .count(slow_count), .tc(tick_10ms));

// in the testbench: the same module, tiny N, so a test takes a few cycles
mod_n_counter #(.N(3), .W(2)) dut (.clk(clk), .reset(reset), .en(en),
                                   .count(count), .tc(tc));
```

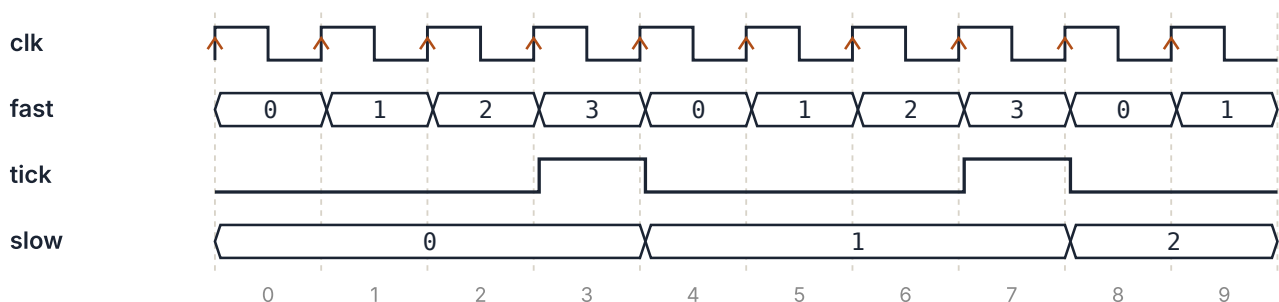
Never edit a constant inside a module to make simulation faster. Override the parameter where you instantiate the module instead; the module that is tested is then the module that goes on the board.

6 Slowing down: enables, not clocks

Often something must happen much more slowly than the 100 MHz board clock: a serial bit every 0.1 s, an LED step every 0.5 s. The tempting idea is to divide the clock and clock a second circuit from the divided signal. **Do not**. Keep every register on `clk` and use a one-cycle **enable** instead.

Two counters, both on `clk`; the first enables the second

```
module two_stage #(parameter integer N1 = 4, parameter integer N2 = 3) (
    input wire    clk, reset,
    output wire [1:0] fast,
    output wire [1:0] slow
);
    wire tick, unused_tc;
    mod_n_counter #(.N(N1), .W(2)) u_fast (.clk(clk), .reset(reset), .en(1'b1),
                                           .count(fast), .tc(tick));
    mod_n_counter #(.N(N2), .W(2)) u_slow (.clk(clk), .reset(reset), .en(tick),
                                           .count(slow), .tc(unused_tc));
endmodule
```



Simulation with $N1 = 4$, $N2 = 3$. `tick` is high for one clock in every four; `slow` advances at the edge that ends that cycle. Both counters share one clock.

Wrong: a logic signal used as a clock

```
always @(posedge tick) // derived clock
    seconds <= seconds + 1;
```

Right: the same behaviour with an enable

```
always @(posedge clk) // the one clock
    if (tick) seconds <= seconds + 1;
```

Why it matters: a clock made by logic arrives later than `clk` (skew), can glitch, and is invisible to the timing analysis that checks your design. Data crossing between the two "clocks" can be captured at the wrong time. With an enable, everything is checked against one clock and the behaviour is the same.

7 Next-value logic and the latch trap

For anything more than a few lines, it is clearer to compute the next value in a combinational block and keep the clocked block trivial. This is also exactly how state machines are written.

A 3-bit up counter that stops at 7

```
module sat_counter (
    input wire      clk, reset, up,
    output reg [2:0] q
);
    reg [2:0] next;
    always @(*) begin
        next = q;           // hold
        if (up && q != 3'd7)
            next = q + 1'b1;
    end
    always @(posedge clk) begin
        if (reset) q <= 3'd0;
        else      q <= next;
    end
endmodule
```

The first block is pure combinational logic: given `q` and `up`, what should the register hold after the next edge? The second block is the register and nothing else.

The line `next = q;` at the top is a **default assignment**. It guarantees that `next` is assigned on every path through the block.

The latch trap. If some path through an `always @(*)` block leaves a variable unassigned, that variable must remember its old value, and the tool builds a level-sensitive **latch**, the circuit from the first lectures that we deliberately avoid. Vivado warns "inferred latch"; treat that warning as an error. Cure: a default assignment at the top of the block, or an `else` on every `if` and a `default` in every `case`. A clocked block never makes a latch: there, "not assigned" means "hold", which is what a flip-flop does.

8 Remembering the past: levels and events

Some signals are **levels**: true for as long as a condition holds ("button is down"). Others are **events**: true for exactly one clock cycle ("button was just pressed"). A controller that treated a level as an event would act again on every cycle the level stayed high. To turn a level into an event, store the level's value from the previous cycle and compare.

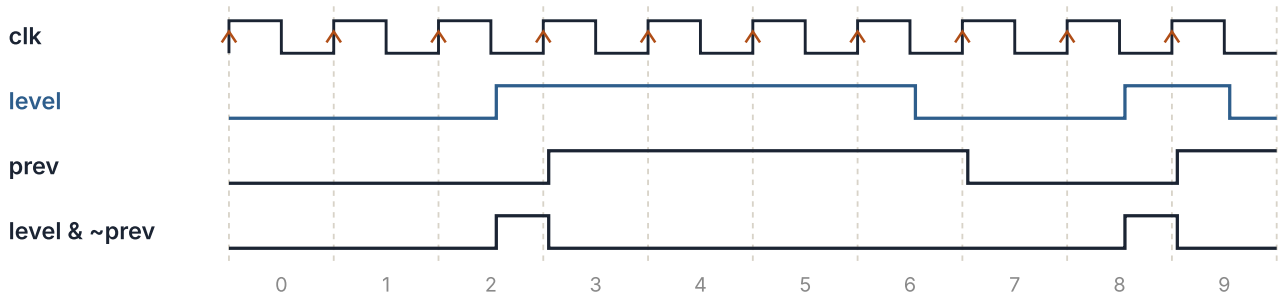
One flip-flop remembers the previous value

```
module prev_value (  
    input wire clk, reset, level,  
    output reg prev  
);  
    always @(posedge clk) begin  
        if (reset) prev <= 1'b0;  
        else      prev <= level;  
    end  
endmodule
```

prev is level delayed by one clock.

Where level is 1 but prev is still 0, the level has just risen.

level & ~prev is therefore 1 for one cycle per rising edge of level, however long the level stays high.



Simulation. The level is high for four cycles, then for one; level & ~prev gives one one-cycle pulse for each. It is combinational, so it rises when level rises and falls at the next clock edge, when prev catches up.

For Lab 6a. The lab's edge detector is built on this idea, but you must also produce a falling-edge pulse and make sure reset does not create a false pulse. Think about what prev should be reset to and why. A real button also needs a synchronizer and a debouncer before this stage: the next section.

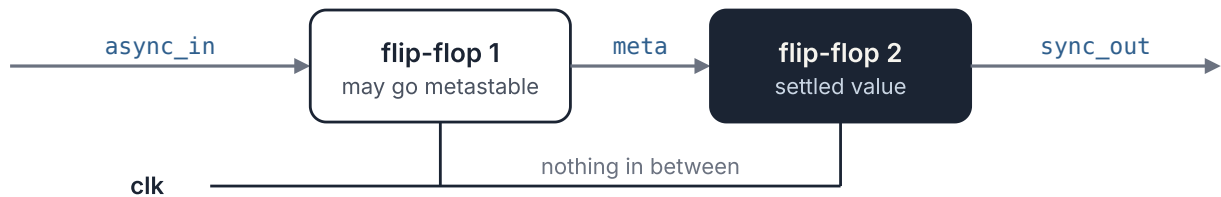
9 Inputs from the outside world (Lab 6a background)

Everything so far assumed that every input changes just after a clock edge, because it comes from another flip-flop on the same clock. A push button does not: a person presses it at any instant, and its metal contacts bounce. Two problems follow, and each has a standard circuit.

9.1 Asynchronous inputs and metastability

Recall from lecture 2: an edge-triggered flip-flop needs its D input steady for a **setup time** before the active edge and a **hold time** after it. Inside a synchronous design the tools check that every path meets this. A button can change *inside* that window. The flip-flop may then go **metastable**: its output hovers between 0 and 1 for an unpredictable time and finally settles to either value. Logic that reads it meanwhile can see a 0 in one place and a 1 in another.

Metastability cannot be prevented, only made extremely unlikely to reach the rest of the circuit. The standard cure is a **two-flip-flop synchronizer**:



Two flip-flops in series on the system clock. If flip-flop 1 goes metastable, it has a whole clock period to settle before flip-flop 2 samples it.

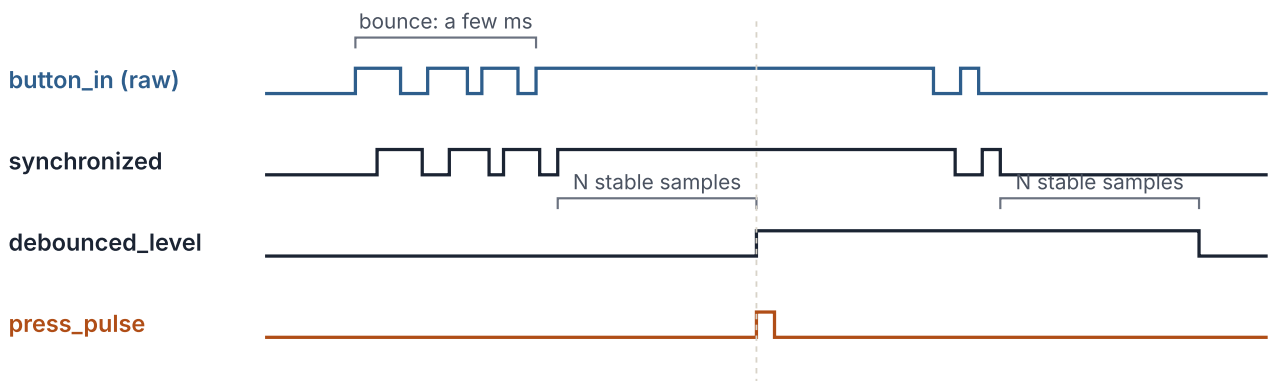
Synchronizer rules

- Exactly two flip-flops back to back, both on `clk`, with **no logic between them**. Logic would eat into the settling time.
- Only the second flip-flop's output is used. Nothing else reads the first.
- Each asynchronous input is synchronized once, at the boundary, and only the synchronized copy is used inside.
- The cost is two clock cycles of delay, which you will see in simulation. Simulation cannot show metastability itself: it only checks the logic and the delay.

9.2 Contact bounce and debouncing

When a mechanical button closes or opens, its contacts bounce for a few milliseconds, making and breaking contact many times. At 100 MHz a few milliseconds is hundreds of thousands of clock cycles, so the synchronized signal shows many separate "presses". A **debouncer** accepts a new value only after it has been stable for a chosen time T_d :

- Keep the currently accepted level (`debounced_level`).
- While the input agrees with it, do nothing and keep the counter at 0.
- While the input disagrees, count consecutive disagreeing samples. If the input agrees again before the count is reached, clear the counter: the qualification starts again from zero.
- When the count reaches N , accept the new value.



Schematic, not to scale: in reality a bounce lasts hundreds of thousands of clock cycles at 100 MHz.

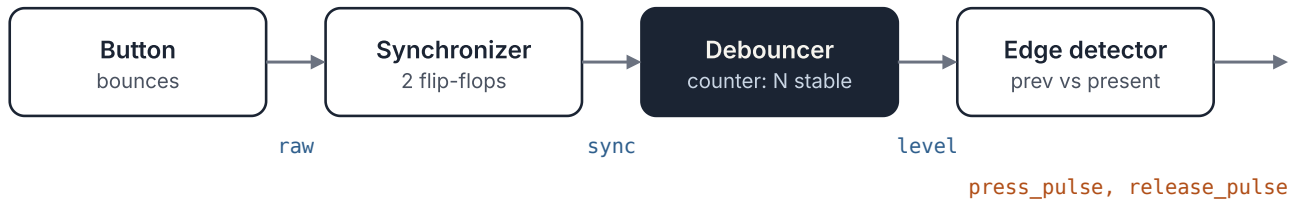
The idea, not cycle-accurate. The raw input bounces; the synchronized copy is the same, two cycles later. The debounced level changes only after the input has been stable for N samples, so each press gives one rising edge and one press pulse.

Sizing N . $N = T_d \times f_{clk}$, and the counter needs $w = \lceil \log_2 N \rceil$ bits. For $T_d = 10$ ms at 100 MHz, $N = 1,000,000$; $2^{20} = 1,048,576$, so 20 bits. In simulation, override N with a small value such as 4 (section

5), so a test takes tens of cycles, not millions.

A debouncer is a small state machine: its state is the accepted level plus the counter value. It is built from parts you already have: a register, a counter with clear, and a comparison.

9.3 The complete input conditioner



The order matters. Synchronize first, so everything after it is ordinary synchronous logic. Debounce before detecting edges, or every bounce becomes an event.

The output is one press event, exactly one clock cycle long, per physical press, however long the button is held. That is what a controller such as the Lab 6b transmitter needs as its `start`: a long level must never be read as many requests.

9.4 Testing it

Model the bouncing button in the testbench. A task like this one produces a few short glitches and then settles; with a small N, check that only the final, stable value is accepted, that exactly one one-cycle pulse comes out, that glitches shorter than N cycles are ignored, and that reset produces no pulse.

A testbench task that imitates a bouncing button

```
task bounce_to(input v); // a few short glitches, then settle at v
begin
    button_in = v; repeat (2) @(negedge clk);
    button_in = ~v; repeat (1) @(negedge clk);
    button_in = v; repeat (1) @(negedge clk);
    button_in = ~v; repeat (2) @(negedge clk);
    button_in = v; // stable from here on
end
endtask
```

10 Testing clocked designs

A testbench is a module with no ports. It creates the clock, drives the inputs, and checks the outputs. Write it so that it **checks itself** and ends with PASS or FAIL: reading waveforms by eye does not scale, and the labs require a pass/fail summary.

```

`timescale 1ns/1ps
module tb_mod_n;
    reg clk = 1'b0, reset = 1'b1, en = 1'b0;
    wire [1:0] count;
    wire tc;
    integer errors = 0;

    mod_n_counter #(.N(3), .W(2)) dut (
        .clk(clk), .reset(reset), .en(en), .count(count), .tc(tc));

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

    task check(input [1:0] exp_count, input exp_tc);
        if (count != exp_count || tc != exp_tc) begin
            $display("FAIL t=%0t count=%0d tc=%b, expected %0d %b",
                $time, count, tc, exp_count, exp_tc);
            errors = errors + 1;
        end
    endtask

    initial begin
        $dumpfile("tb_mod_n.vcd"); $dumpvars(0, tb_mod_n);
        @(negedge clk); reset = 1'b0;      // release reset between edges
        check(2'd0, 1'b0);
        en = 1'b1;                          // count from the next edge
        @(negedge clk) check(2'd1, 1'b0);
        @(negedge clk) check(2'd2, 1'b1);   // tc high in the last state, N-1
        @(negedge clk) check(2'd0, 1'b0);   // wrapped back to 0
        @(negedge clk) check(2'd1, 1'b0);
        en = 1'b0;
        @(negedge clk) check(2'd1, 1'b0);   // en = 0: count holds
        if (errors == 0) $display("PASS");
        else $display("FAIL: %0d errors", errors);
        $finish;
    end
endmodule

```

Technique	Why
<code>always #5 clk = ~clk;</code>	A free-running clock: period 10 ns with <code>`timescale 1ns/1ps</code> .
Change inputs at <code>@(negedge clk)</code>	Inputs then change half a cycle away from the rising edge where the design samples them, so there is no race between the testbench and the design.
Check just after the negative edge	Outputs have settled after the rising edge. One <code>@(negedge clk)</code> = one clock cycle of the design.
<code>!==</code> , not <code>!=</code>	<code>!==</code> also compares <code>x</code> and <code>z</code> . With <code>!=</code> , an output stuck at <code>x</code> would make the test silently pass.
A <code>task</code> for the check	One line per expected value; failures print the time and both values.
<code>N = 3</code> , not 4	Choose test sizes that exercise every path. With <code>N = 4</code> and <code>W = 2</code> the counter wraps from 3 to 0 by itself, so a missing "clear at <code>N-1</code> " would go unnoticed. With <code>N = 3</code> the same bug fails three checks.
<code>\$dumpfile</code> , <code>\$dumpvars</code>	Write a waveform file you can open when a check fails.
Test every behaviour the specification names: reset, every transition, holding when not enabled, wrap-around, one-cycle outputs that must not last two cycles, and reset in the middle of operation.	

11 Rules and common mistakes

Mistake	What goes wrong, and the fix
<code>=</code> in a clocked block	Order-dependent behaviour; flip-flops disappear (section 3). Use <code><=</code> .
Same <code>reg</code> assigned in two blocks	Multiple drivers: simulation and hardware disagree, synthesis fails. One block per variable.
Missing assignment in <code>@(*)</code>	Inferred latch (section 7). Default assignments first.
<code>always @(a or b)</code> with a missing signal	Simulation ignores changes on the missing input; hardware does not. Use <code>@(*)</code> .
<code>@(posedge some_signal)</code> on a data signal	A derived clock (section 6). Use <code>@(posedge clk)</code> and an enable.
No reset, or reset only in the testbench	Registers start as <code>x</code> in simulation and random on the board. Reset every register in the design.
<code>initial</code> or <code>#</code> delays in a design module	Not synthesizable as you intend; ignored or rejected. Only in testbenches.
Counter too narrow	Wraps before reaching $N-1$; the terminal count never comes. $W \geq \lceil \log_2 N \rceil$.
Inputs changed at <code>posedge clk</code> in the testbench	A race: the design may see the old or the new value. Drive at <code>negedge</code> .
Checking with <code>!=</code>	<code>x</code> outputs pass unnoticed. Use <code>!==</code> .

12 Exercises

1. What hardware does this describe? What are `x`, `y` and `z` one clock after `d` changes?

```
always @(posedge clk) begin
    x = d;
    y = x;
    z = y;
end
```

2. Why does this block build a latch? Rewrite it so that `y` is `b` when `sel` is 0.

```
always @(*)
    if (sel) y = a;
```

3. Write an 8-bit register with synchronous `reset`, `load` (load `d`) and `shift` (shift left, filling with 0). If both `load` and `shift` are 1, `load` wins.

4. The Basys3 clock is 100 MHz. Make an LED blink at 1 Hz (on for 0.5 s, off for 0.5 s) using the `mod_n_counter` from section 5 and no derived clock. What `N` and `W` do you need? How would you test it in simulation?

5. In the testbench of section 10, suppose the line `en = 1'b1;` is moved down one line, to just after `@(negedge clk) check(2'd1, 1'b0);`. Which checks fail, and why?

6. A button is to be debounced with $T_d = 20$ ms on the 100 MHz board clock. What is `N`, and how many bits does the counter need?

7. Why must the synchronizer come before the debouncer, and the debouncer before the edge detector? What goes wrong in each case if the order is swapped?

Answers

1. Blocking assignments execute in order, so at each edge `x` gets `d`, then `y` gets the new `x`, then `z` the new `y`: all three equal `d` after one clock. It is one flip-flop with three names, not a three-stage shift register. With `<=` it would be three flip-flops in a chain (section 3).

2. When `sel` is 0, `y` is not assigned, so it must hold its old value: a latch. Fix:

```
always @(*)
    if (sel) y = a;
    else    y = b;      // or: y = b; if (sel) y = a;
```

3.

```

module ld_shift (
    input wire    clk, reset, load, shift,
    input wire [7:0] d,
    output reg [7:0] q
);
    always @(posedge clk) begin
        if (reset)      q <= 8'd0;
        else if (load)  q <= d;
        else if (shift) q <= {q[6:0], 1'b0};
    end
endmodule

```

The `if/else if` order gives the priority. Checked in simulation: with `load` and `shift` both 1, `d` is loaded; one shift later 1011_0011 becomes 0110_0110.

4. The LED must toggle every 0.5 s = 50,000,000 clock cycles, so $N = 50,000,000$. $2^{25} = 33,554,432$ is too small and $2^{26} = 67,108,864$ is enough, so $W = 26$.

```

module blinker #(parameter integer N = 50_000_000, parameter integer W = 26) (
    input wire clk, reset, output reg led);
    wire tick;
    wire [W-1:0] unused_count;
    mod_n_counter #(.N(N), .W(W)) u_div (.clk(clk), .reset(reset), .en(1'b1),
                                         .count(unused_count), .tc(tick));

    always @(posedge clk)
        if (reset)      led <= 1'b0;
        else if (tick) led <= ~led;
endmodule

```

To test it, instantiate `blinker #(.N(5), .W(3))` and check that `led` toggles every 5 cycles (6 toggles in 30 cycles in our simulation). The same module with the default parameters goes on the board.

5. All five checks after the first. The counter now stays at 0 for one extra cycle, so it is one step behind every expected value: 0 instead of 1, 1 instead of 2 (and `tc` = 0 instead of 1), and so on; the last check fails too, because counting stopped at 0. Our simulation prints five FAIL lines. A testbench that fails loudly when the stimulus is off by one cycle is doing its job.

6. $N = 0.020 \text{ s} \times 100,000,000 \text{ Hz} = 2,000,000$. $2^{20} = 1,048,576$ is too small and $2^{21} = 2,097,152$ is enough, so the counter needs 21 bits.

7. The debouncer is ordinary synchronous logic: its counter and comparison assume an input that changes only just after clock edges. Feeding it the raw button would let a metastable or mid-cycle change reach several flip-flops at once, which could then disagree. The edge detector compares this cycle with the last; fed with a bouncing level, it would produce a pulse for every bounce, many "presses" for one physical press.