

LECTURE 8 · FINITE STATE MACHINES

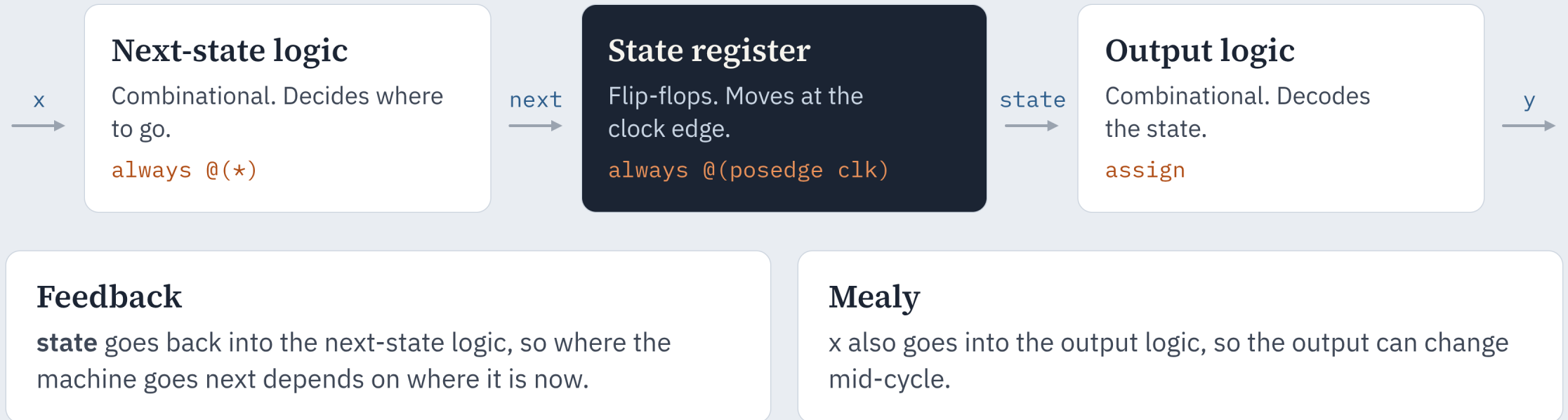
State Machines in Verilog

A template for every FSM, how to test one, and a controller that runs a datapath.

Based on Mano & Ciletti, Digital Design (6th ed.), Section 5.6 (synthesizable HDL models of sequential circuits); examples written for this course and simulated with Icarus Verilog

THE TEMPLATE

Every state machine has three blocks



One piece of code for each block. Keep them apart and the code reads like the diagram.

STEP 3 IN CODE

Name the states, then choose the codes

```
localparam [1:0] S0 = 2'b00, S1 = 2'b01,  
                S2 = 2'b10, S3 = 2'b11;  
reg [1:0] state, next;
```

Named states

Write S2, not 2'b10. Step 3, the encoding, now lives in one place.

localparam

A constant inside the module. Unlike a parameter, it cannot be overridden from outside.

Two registers?

No: **state** is the flip-flops; **next** is a wire-like variable from @(*).

BLOCK 1

The state register

```
always @(posedge clk)
  if (reset) state <= S0;
  else      state <= next;
```

The only clocked block. It never changes from one FSM to the next.

Synchronous, active-high **reset** to the start state.

Nonblocking **<=**, as in every clocked block.

All the thinking goes into the next-state block. This one is copied.

BLOCK 2

Next-state logic: the diagram as a case statement

```
always @(*)
  case (state)
    S0:    next = x ? S1 : S0;
    S1:    next = x ? S2 : S0;
    S2:    next = x ? S3 : S0;
    S3:    next = x ? S3 : S0;
    default: next = S0;
  endcase
```

One case item per state: **the arrows leaving that circle.**

@(*) and blocking =: combinational logic.

default sends unused codes back to S0. Here all four codes are used, but write it anyway.

A TRAP

A missing assignment makes a latch

Wrong: when $x = 0$, next is not assigned

```
always @(*)
  case (state)
    S0: if (x) next = S1;
    S1: if (x) next = S2;
    ...
  endcase
```

Right: every path assigns next

```
always @(*) begin
  next = state;    // default first
  case (state)
    S0: if (x) next = S1;
    ...
  endcase
end
```

If some path through $@(*)$ leaves a variable unassigned, it must *remember* its old value. The tool builds a **latch**.

BLOCK 3

Output logic: Moore or Mealy

Moore

```
assign y = (state == S3);
```

Uses state only. Changes only just after a clock edge.

Mealy

```
assign z = (state == S2) && x;
```

Uses state and input. Follows x within the cycle.

With several outputs, write one assign each, or a second @(*) block with a default value for every output first.

EXAMPLE 1

The three-1s detector, complete

```
module ones3_moore (  
    input wire clk, reset, x,  
    output wire y  
);  
    localparam [1:0] S0 = 2'b00, S1 = 2'b01,  
                    S2 = 2'b10, S3 = 2'b11;  
    reg [1:0] state, next;  
  
    always @(posedge clk)  
        if (reset) state <= S0;  
        else      state <= next;
```

```
    always @(*)  
        case (state)  
            S0:      next = x ? S1 : S0;  
            S1:      next = x ? S2 : S0;  
            S2:      next = x ? S3 : S0;  
            S3:      next = x ? S3 : S0;  
            default: next = S0;  
        endcase  
  
    assign y = (state == S3);  
endmodule
```

Same machine as Mano Fig. 5.27. The tool finds the equations; it need not find ours.

CODE TO CIRCUIT

From the case statement to gates

case item	state A B	next if x = 0	next if x = 1
S0	0 0	0 0	0 1
S1	0 1	0 0	1 0
S2	1 0	0 0	1 1
S3	1 1	0 0	1 1

Each **case** line is one row of the state table, with the codes from the **localparam**.

next[1] is DA. It is 1 only when $x = 1$, in S1, S2 or S3 (A or B is 1):

$$DA = x(A + B)$$

next[0] is DB. It is 1 only when $x = 1$, in S0, S2 or S3 (A = 1 or B = 0):

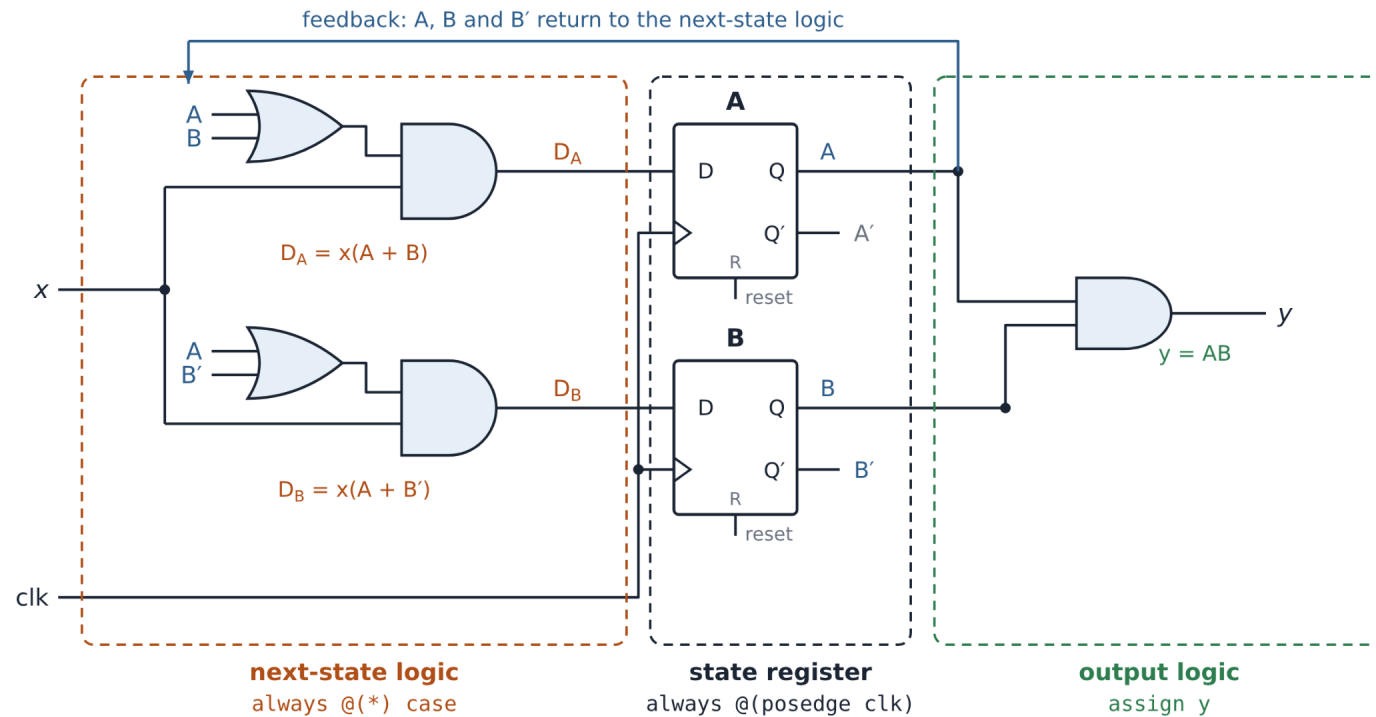
$$DB = x(A + B')$$

assign y = (state == S3) compares both bits: $y = AB$.

The same equations as the K-maps of Lecture 7.

EXAMPLE 1 · HARDWARE

What the three-1s module builds



reg [1:0] state → two D flip-flops, A and B.

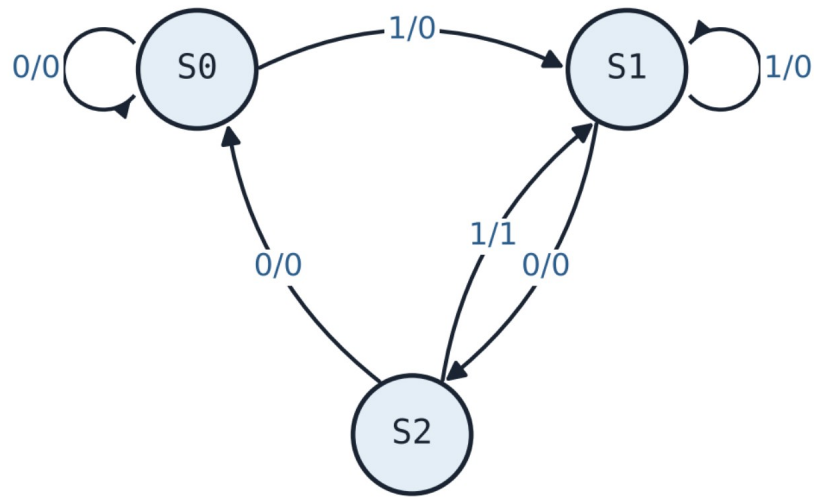
case → the gates in front of D.

assign y → one AND gate after the flip-flops.

The loop from Q back to the gates makes the next state depend on the present state: that is how it counts a run of 1s.

RECAP

The 101 detector from Lecture 7



S0 nothing useful · **S1** last input 1 · **S2** last two inputs 1 0.

Codes S0 = 00, S1 = 01, S2 = 10. Code **11 is unused**.

By hand, with 11 as a don't-care: $z = Ax$. In state 11 with $x = 1$ that gives a **false detection**.

Watch how the Verilog version avoids it.

EXAMPLE 2

The 101 detector, Mealy

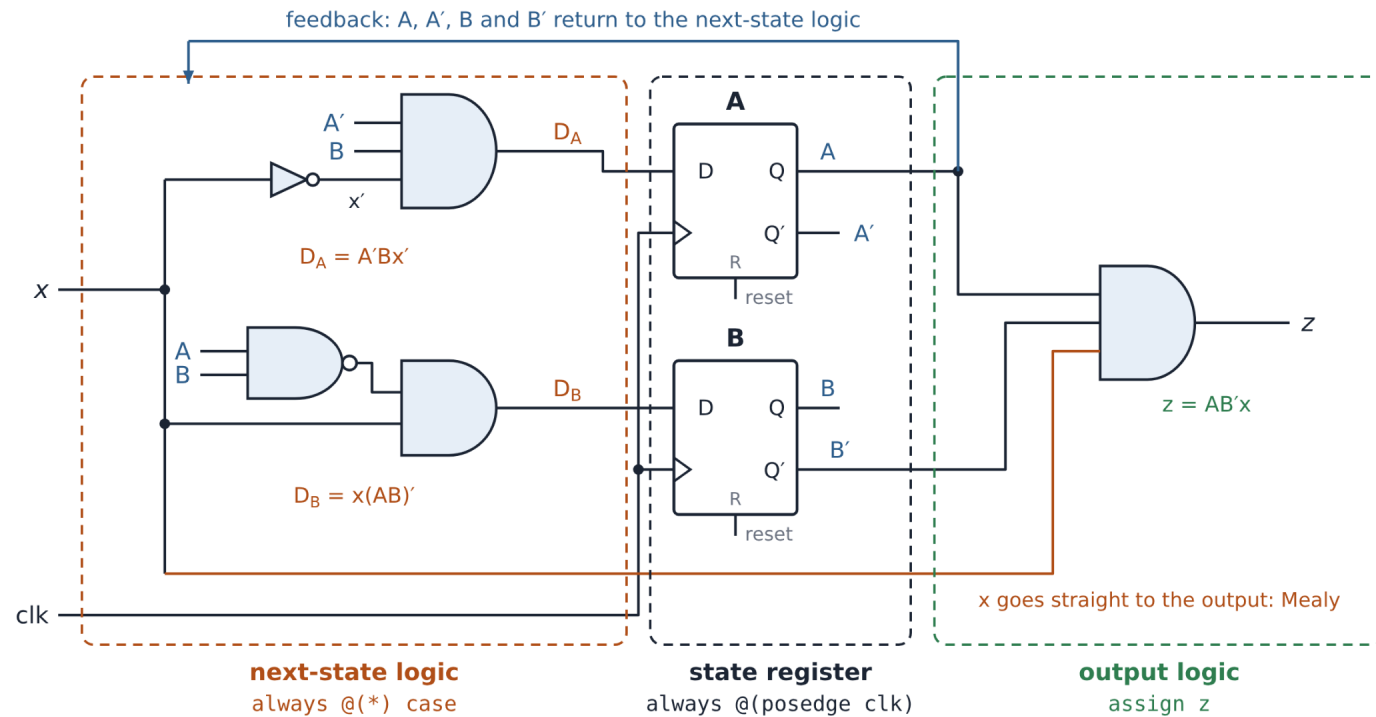
```
module det101_mealy (  
    input  wire clk, reset, x,  
    output wire z  
);  
    localparam [1:0] S0 = 2'd0, S1 = 2'd1,  
                   S2 = 2'd2;    // 2'd3 unused  
    reg [1:0] state, next;  
  
    always @(posedge clk)  
        if (reset) state <= S0;  
        else      state <= next;
```

```
    always @(*)  
        case (state)  
            S0:      next = x ? S1 : S0;  
            S1:      next = x ? S1 : S2;  
            S2:      next = x ? S1 : S0;  
            default: next = S0;  
        endcase  
  
    assign z = (state == S2) && x;  
endmodule
```

The **default** sends code 11 to S0. And $z = (\text{state} == S2) \ \&\& \ x$ is the safe output $z = AB'x$: no false 1 in state 11.

EXAMPLE 2 · HARDWARE

What the 101 module builds



x reaches z **without** passing a flip-flop: that wire is what Mealy means.

By hand, with 11 as don't-care: $DA = Bx'$, $DB = x$, $z = Ax$.

The code says $11 \rightarrow S0$ and z only in $S2$, so the tool must build a little more: A' in DA , $(AB)'$ in DB , B' in z .

A few more gate inputs buy a machine that is safe in state 11.

ON THE FPGA

What Vivado actually builds

module	flip-flops	LUTs	one LUT computes
three-1s detector	2	up to 3	DA, DB, y
101 detector	2	up to 3	DA, DB, z

No AND or OR gates. A LUT is a small memory holding a truth table of up to 6 inputs. Any of our equations fits in one.

Flip-flops are FDRE cells: D, clock, clock enable, and a synchronous reset pin R. reset goes straight to R.

The schematic in Vivado shows LUTs and FDREs, not gates, and the tool may re-encode the states (for example one-hot). The behaviour is the same.

CLEAN OUTPUTS

Registering an output

```
reg z_r;  
always @(posedge clk)  
    if (reset) z_r <= 1'b0;  
    else      z_r <= (state == S2) && x;
```

	1	2	3	4	5	6
x	1	0	1	0	1	1
z (Mealy)	0	0	1	0	1	0
z_r	0	0	0	1	0	1

Combinational output

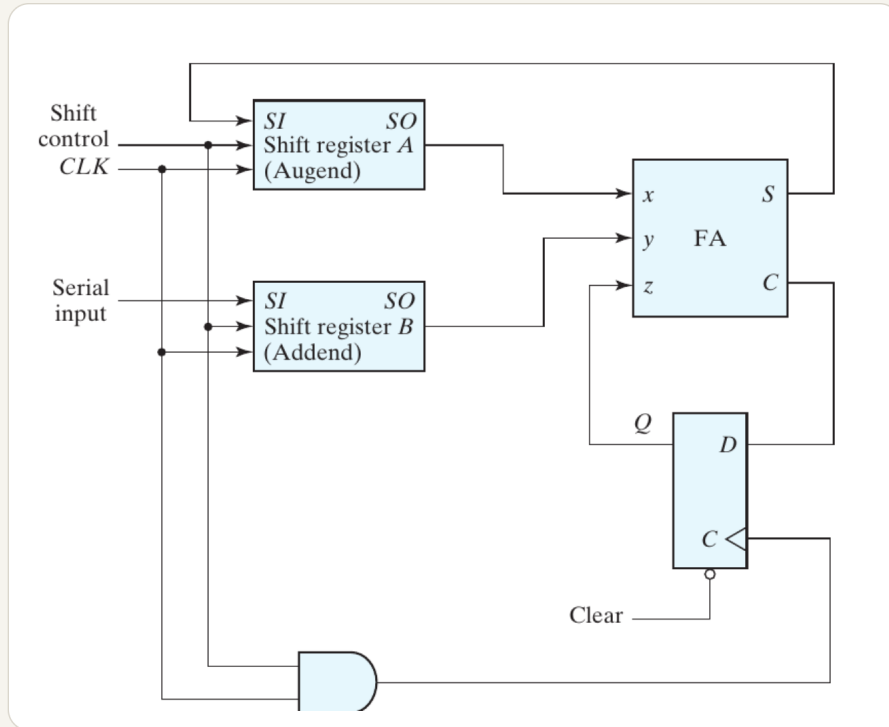
Fast, but any glitch or late change on x reaches the next module.

Registered output

One cycle later, but clean and stable for the whole cycle.
Use it for outputs that leave the module.

RECALL • LECTURE 3

The serial adder we built was a datapath



Shift registers, a full adder, a carry flip-flop: it **stores and computes**, but decides nothing.

Lecture 3 ended with two questions it could not answer: who **clears the carry** first, and who **stops shifting** after exactly n clocks?

Shift control was an input that something else had to drive.

Today: this circuit becomes the **datapath**. We add an **FSM controller** that drives it, plus load and a bit counter so the controller can do its job.

When to split a design

One FSM is enough

The detectors remember very little: a run of up to three 1s, or the last two inputs.

4 states and 3 states. All the memory is the state.

Split it

The serial adder holds A, B, a carry and a count: $16 \times 16 \times 2 \times 4 = \mathbf{2,048}$ combinations as one FSM.

Split: registers, an adder and a counter hold the data; the controller needs **3** states.

Rule: if the state would have to include data values or long counts, put them in a **datapath**. The **controller** only decides what happens next.

In the labs: the Lab 6b transmitter and Lab 6c receiver are a small FSM controlling a shift register and a bit counter.

CONTROLLER AND DATAPATH

Split the design: who decides, who does

Controller

A small FSM: IDLE, ADD, DONE

Inputs: **start**, status

Outputs: control signals, **busy**, **done**

load

shift

last

Datapath

Registers A and B

Full adder and carry flip-flop

Bit counter

Data in: a_in, b_in · out: sum

Control signals

Controller → datapath: what to do on the next edge.

Status signals

Datapath → controller: facts the controller decides on.

DATAPATH

Plan the datapath as a table first

Register	on load	on shift	otherwise
A	a_in	{s, A[N-1:1]}	hold
B	b_in	B >> 1	hold
c (carry)	0	carry out	hold
cnt	0	cnt + 1	hold

Full adder

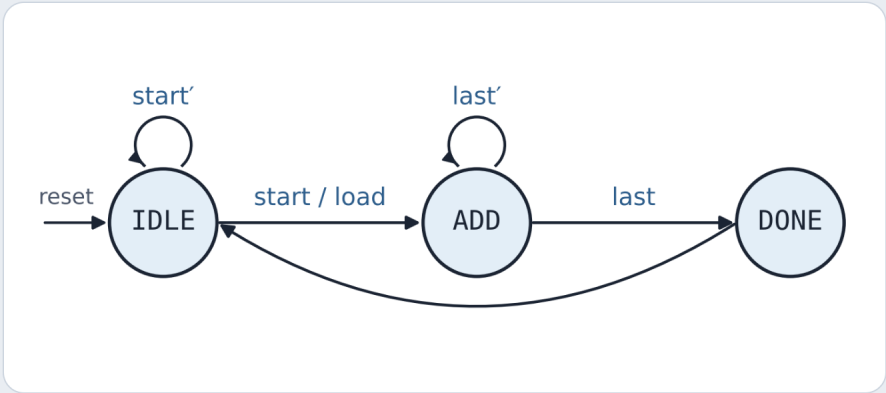
$s = A0 \oplus B0 \oplus c$, carry out from A0, B0, c.

Status

last = (cnt == N-1): this is the final bit.

CONTROLLER

The controller: three states



Inputs **start** and **last** choose the arrows. **load** is a Mealy output: it depends on start. The others are Moore.

Next state

State	Input	Next
IDLE	start = 0	IDLE
IDLE	start = 1	ADD
ADD	last = 0	ADD
ADD	last = 1	DONE
DONE	–	IDLE

Outputs in each state

State	load	shift	busy	done
IDLE	start	0	0	0
ADD	0	1	1	0
DONE	0	0	0	1

TIMING

One addition, cycle by cycle

Cycle	state	start	load	shift	cnt	last	A	c	done
0	IDLE	1	1	0	–	–	–	–	0
1	ADD	0	0	1	0	0	0101	0	0
2	ADD	0	0	1	1	0	0010	1	0
3	ADD	0	0	1	2	0	0001	1	0
4	ADD	0	0	1	3	1	0000	1	0
5	DONE	0	0	0	0	0	1000	0	1
6	IDLE	0	0	0	0	0	1000	0	0

$N = 4$, $a_{in} = 0101$ (5), $b_{in} = 0011$ (3). Sum 1000 (8) is ready in cycle 5, $N + 1$ cycles after start.

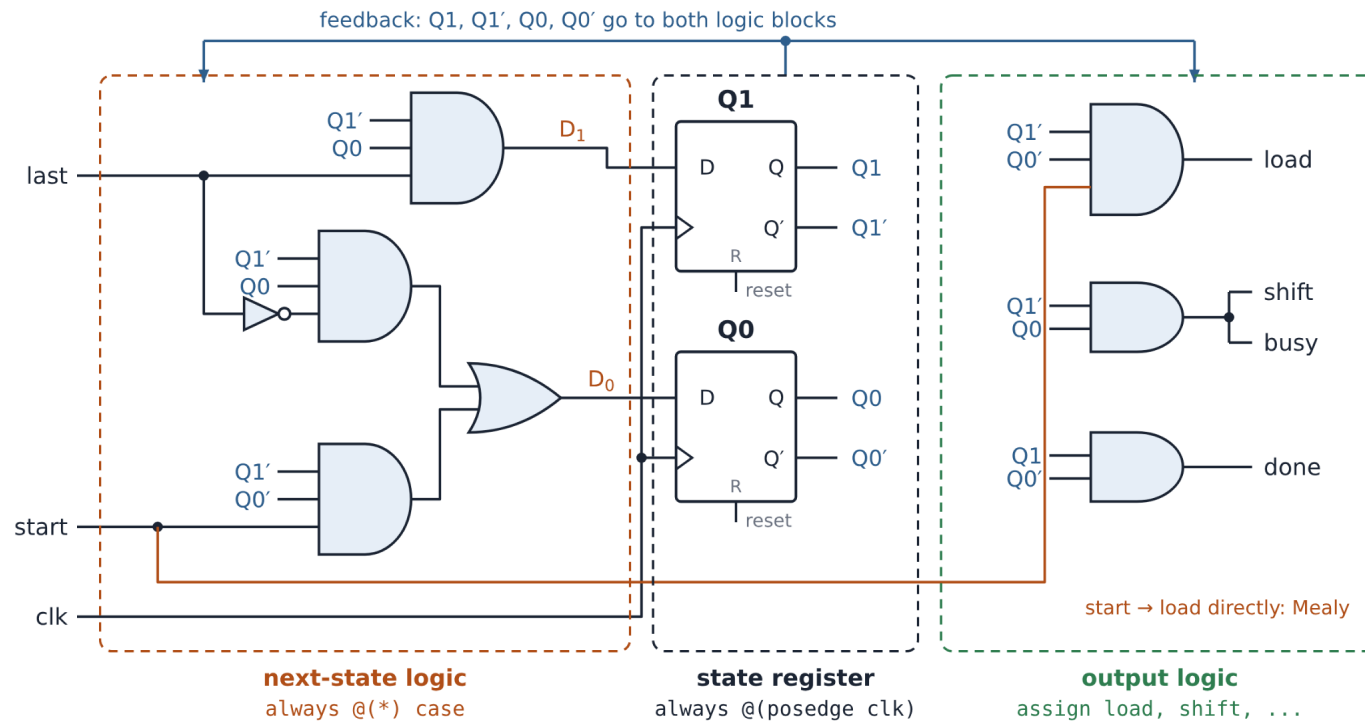
CONTROLLER CODE

The controller: the same three blocks

```
module sa_control (  
    input  wire clk, reset, start, last,  
    output wire load, shift, busy, done  
);  
    localparam [1:0] IDLE = 2'd0, ADD = 2'd1,  
                   DONE = 2'd2;  
    reg [1:0] state, next;  
  
    always @(posedge clk)  
        if (reset) state <= IDLE;  
        else      state <= next;
```

```
    always @(*)  
        case (state)  
            IDLE:    next = start ? ADD  : IDLE;  
            ADD:     next = last  ? DONE : ADD;  
            DONE:    next = IDLE;  
            default: next = IDLE;  
        endcase  
  
    assign load  = (state == IDLE) && start;  
    assign shift = (state == ADD);  
    assign busy  = (state == ADD);  
    assign done  = (state == DONE);  
endmodule
```

What sa_control builds



IDLE = 00, ADD = 01, DONE = 10: two flip-flops, $Q1$ $Q0$.

$$D_1 = Q1'Q0 \cdot last$$

$$D_0 = Q1'Q0' \cdot start + Q1'Q0 \cdot last'$$

$$load = Q1'Q0' \cdot start$$

$$shift = busy = Q1'Q0$$

$$done = Q1Q0'$$

The same three boxes as the detectors. Only the inputs (*start*, *last*) and outputs differ.

DATAPATH CODE

The datapath: registers and a full adder

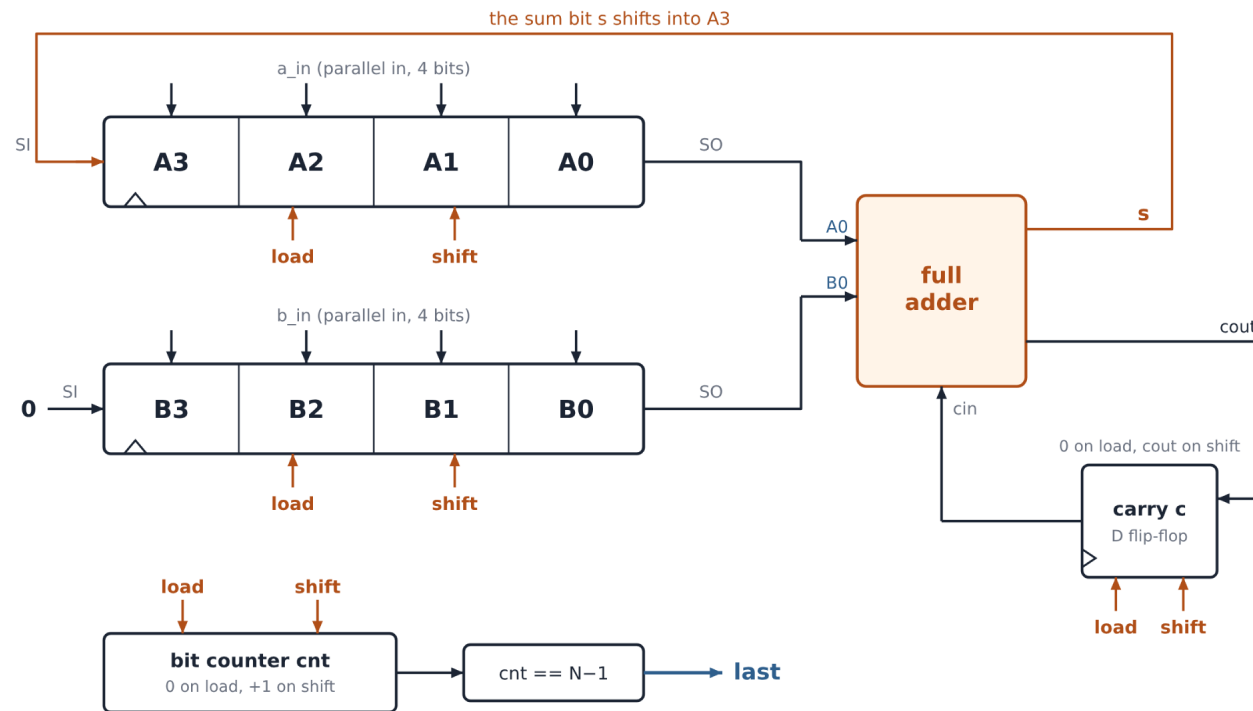
```
module sa_datapath #(parameter integer N = 4) (  
    input  wire      clk, load, shift,  
    input  wire [N-1:0] a_in, b_in,  
    output reg [N-1:0] a,  
    output wire      last  
);  
    localparam integer W = $clog2(N);  
    reg [N-1:0] b;  
    reg      c;  
    reg [W-1:0] cnt;  
    wire s      = a[0] ^ b[0] ^ c;  
    wire cout = (a[0] & b[0]) | (c & (a[0] ^ b[0]));  
    assign last = (cnt == N-1);
```

```
always @(posedge clk)  
    if (load) begin  
        a <= a_in;  b <= b_in;  
        c <= 1'b0;  cnt <= 0;  
    end else if (shift) begin  
        a <= {s, a[N-1:1]};  
        b <= b >> 1;  
        c <= cout;  
        cnt <= cnt + 1'b1;  
    end  
endmodule
```

One clocked block, one branch per row of the datapath table. No state, no decisions.

DATAPATH · HARDWARE

What sa_datapath builds ($N = 4$)



load and shift (orange) come from the controller; all flip-flops share clk

A and B: **shift registers with parallel load**, as in Lecture 3. load: take a_in, b_in. shift: shift right. Neither: hold.

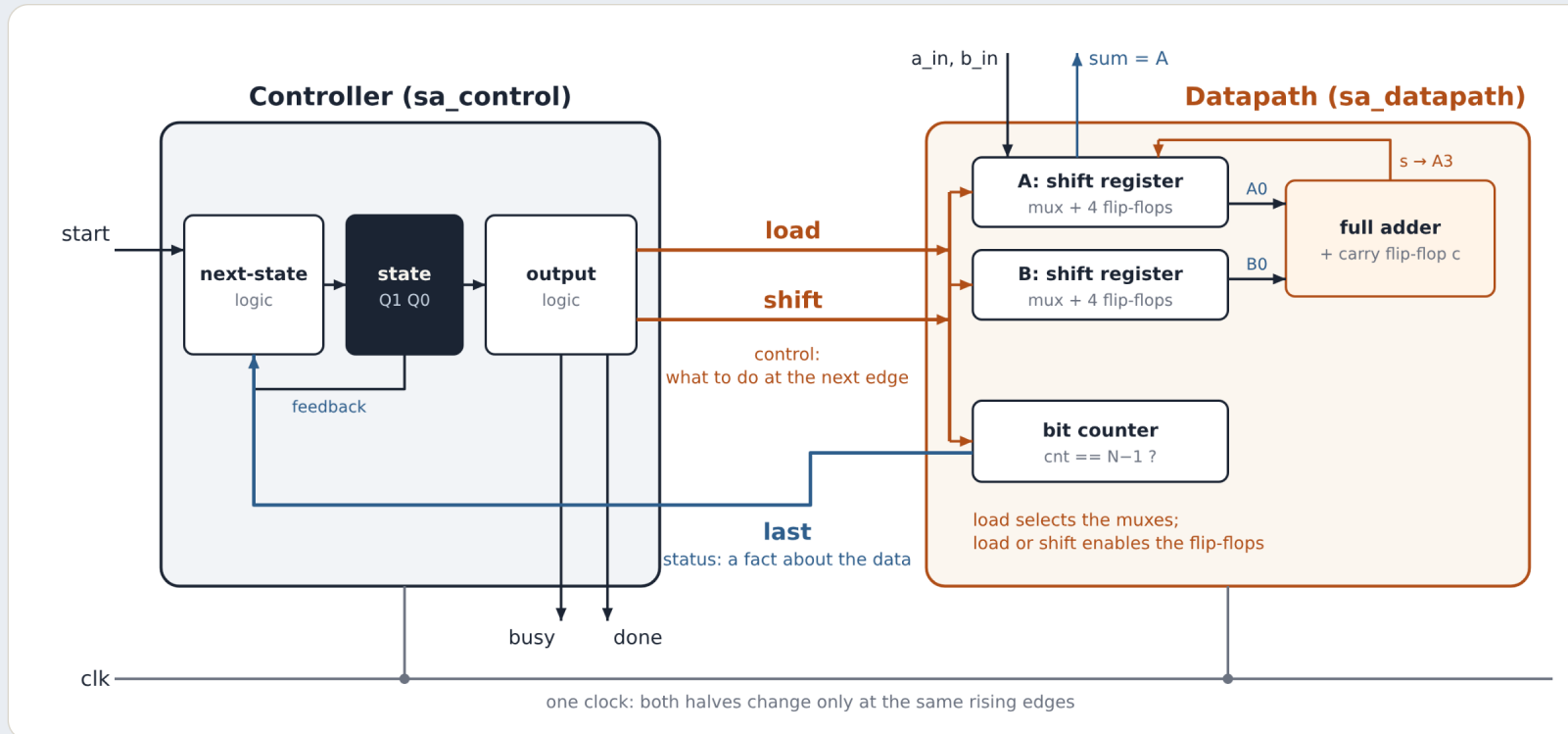
Mano's serial adder, Fig. 6.5, plus three things: load, a carry cleared on load, and a bit counter.

One full adder, used once per clock.

last = (cnt == $N-1$) goes back to the controller.

CONTROLLER + DATAPATH

How the two halves talk



load, shift: the controller tells the datapath what to do at the next edge.

last: the datapath tells the controller a fact. It changes the controller's next state, ADD → DONE.

One clock: both halves act at the same edge.

CONTROLLER + DATAPATH

One addition: who tells whom

Cycle	State	Controller → datapath	Datapath, at the edge	Datapath → controller	Controller, at the same edge
0	IDLE	load = 1	loads A, B; c, cnt ← 0	–	IDLE → ADD (start)
1–3	ADD	shift = 1	adds one bit, shifts, cnt + 1	last = 0	stays in ADD
4	ADD	shift = 1	adds the last bit	last = 1	ADD → DONE
5	DONE	none	holds: sum is ready	–	DONE → IDLE; done = 1

At the end of cycle 4, the datapath does its last shift **and** the controller leaves ADD, at the same edge. last arrived just in time, during cycle 4.

Neither half works alone: the datapath only obeys load and shift; the controller never sees the numbers, only last.

PUTTING IT TOGETHER

The serial adder unit: two instances, three wires

```
module serial_adder_unit #(parameter integer N = 4) (  
    input wire      clk, reset, start,  
    input wire [N-1:0] a_in, b_in,  
    output wire [N-1:0] sum,  
    output wire      busy, done  
);  
    wire load, shift, last;    // the arrows between the halves  
  
    sa_control ctl (.clk(clk), .reset(reset), .start(start),  
                   .last(last), .load(load), .shift(shift),  
                   .busy(busy), .done(done));  
  
    sa_datapath #(N) dp (.clk(clk), .load(load), .shift(shift),  
                        .a_in(a_in), .b_in(b_in),  
                        .a(sum), .last(last));  
  
endmodule
```

start while idle: load a_in and b_in, clear the carry. Then one bit per clock, N clocks, LSB first.

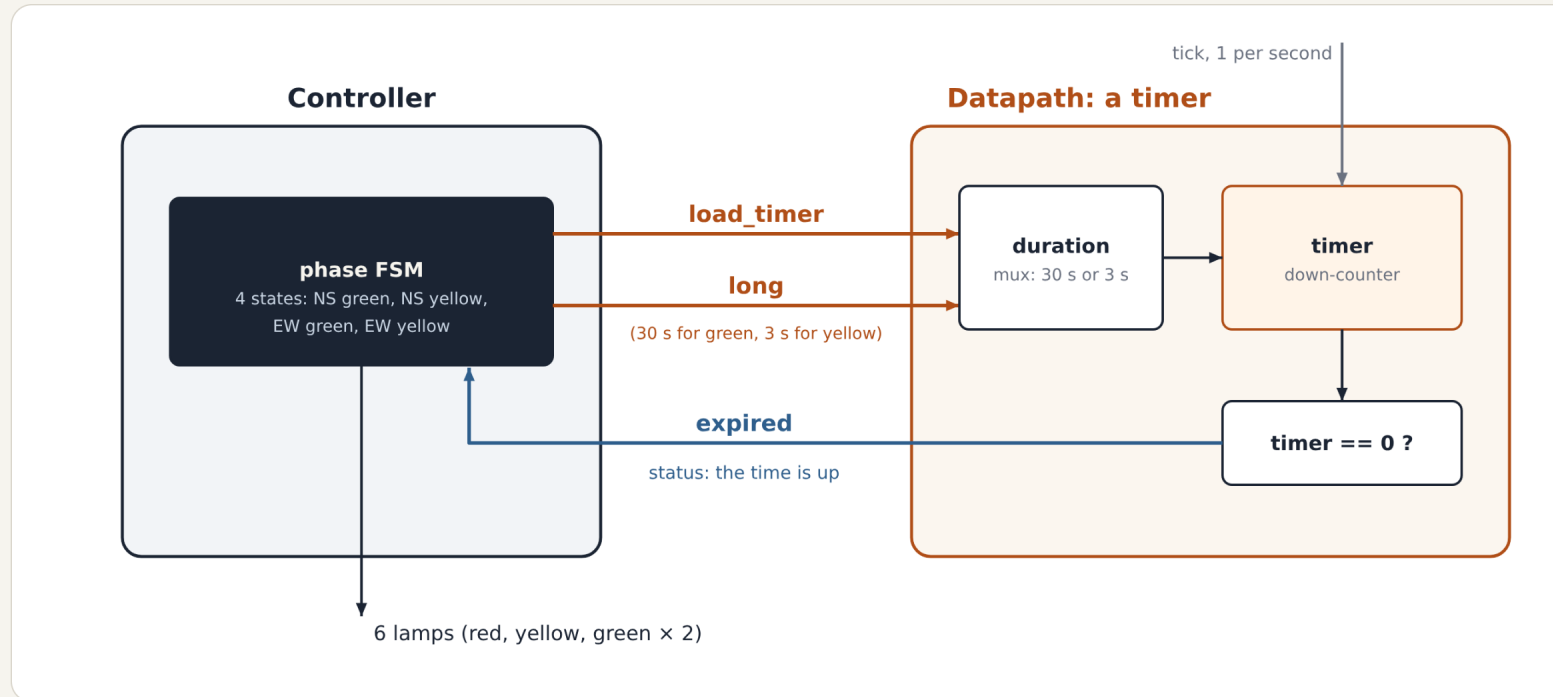
busy = 1 while adding; start is ignored. **done** = 1 for one cycle; then $\text{sum} = a_in + b_in \pmod{2^n}$.

load, shift, last are internal wires, not ports: they join the two instances.

ctl and **dp** are instances: copies of the modules. **.a(sum)** makes register A the output.

MORE EXAMPLES · TRAFFIC LIGHT

A traffic light: phases and a timer



The **controller** remembers the phase: 4 states.

The **timer** remembers the time. 30 s is 3 billion clocks: impossible as states, easy as a counter that counts a 1-second tick.

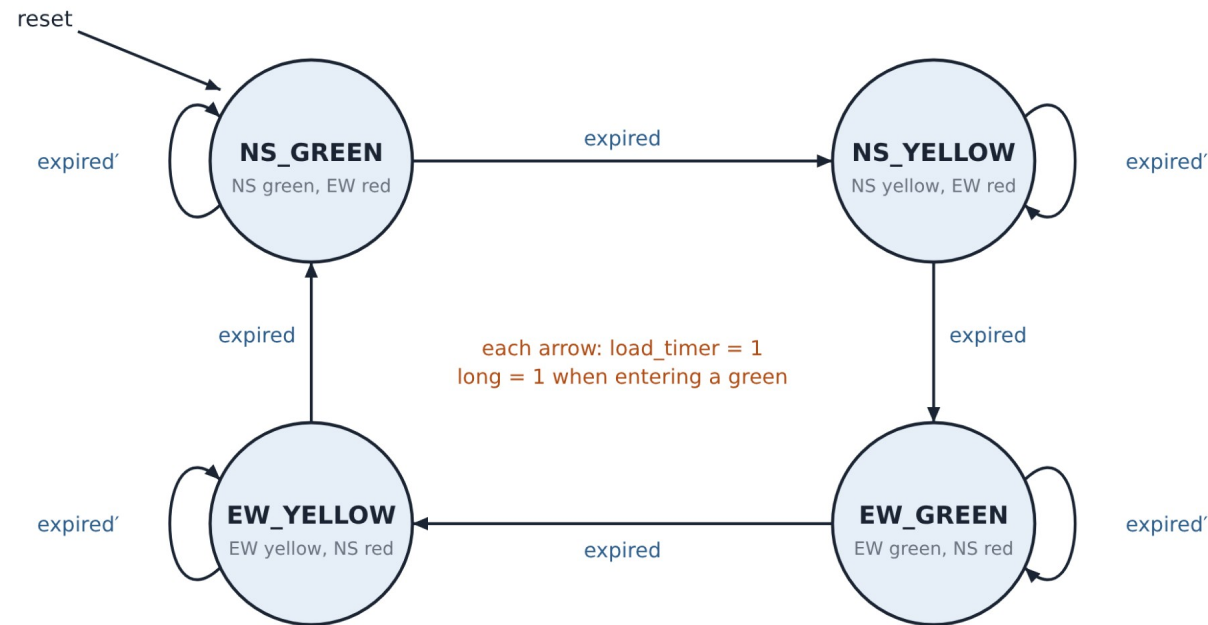
Control: **load_timer**, **long**.

Status: **expired**.

The serial adder again: load/shift → load_timer; last → expired.

MORE EXAMPLES · TRAFFIC LIGHT

The traffic-light controller



Every arrow is taken when expired = 1; otherwise the light stays.

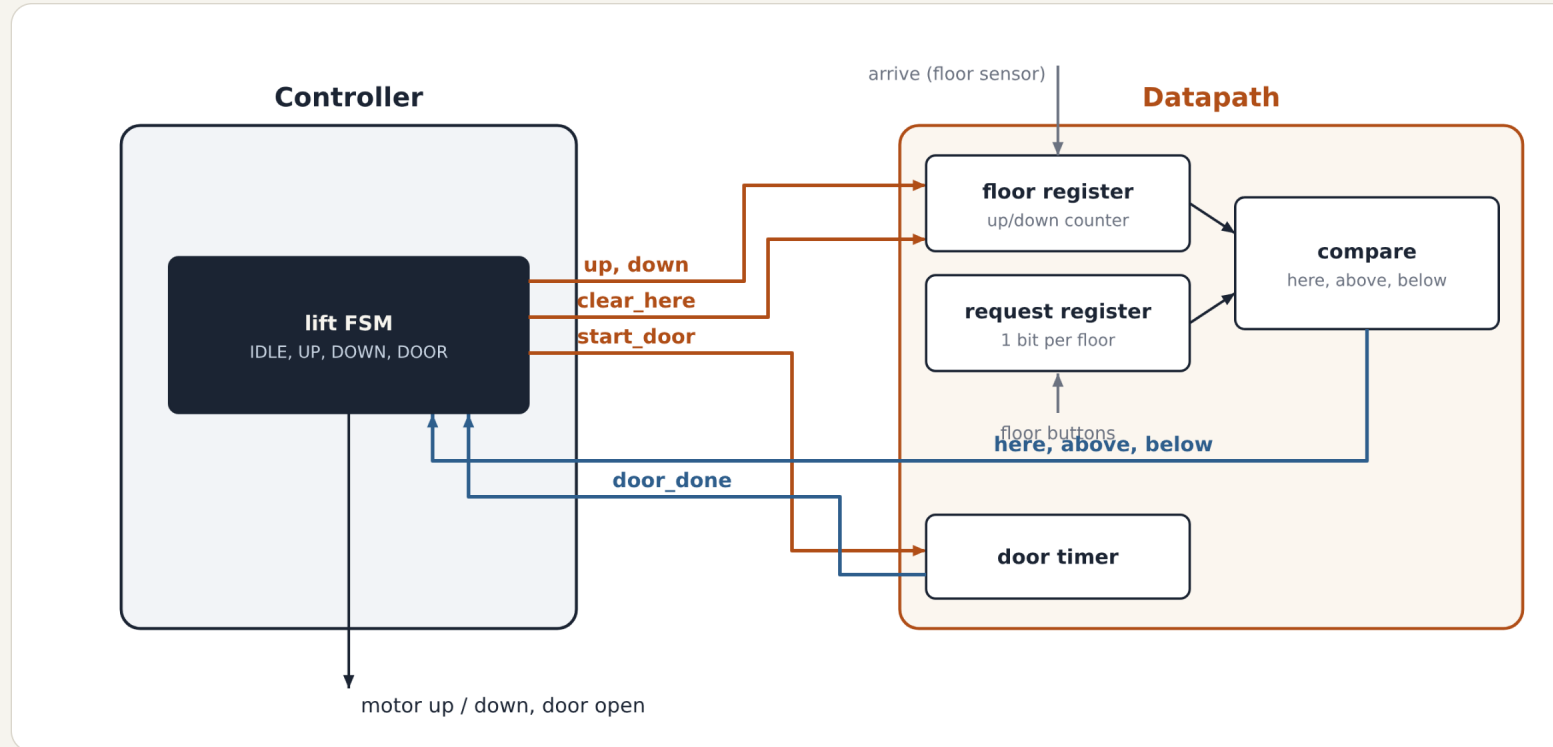
load_timer = expired: reload on every change of phase.

long = 1 in the yellow states, because the next phase is a green.

The lamps depend on the state only:
Moore outputs.

MORE EXAMPLES · LIFT

A lift: a small FSM, a lot of data



Data: the current floor, one request bit per floor, the door time.

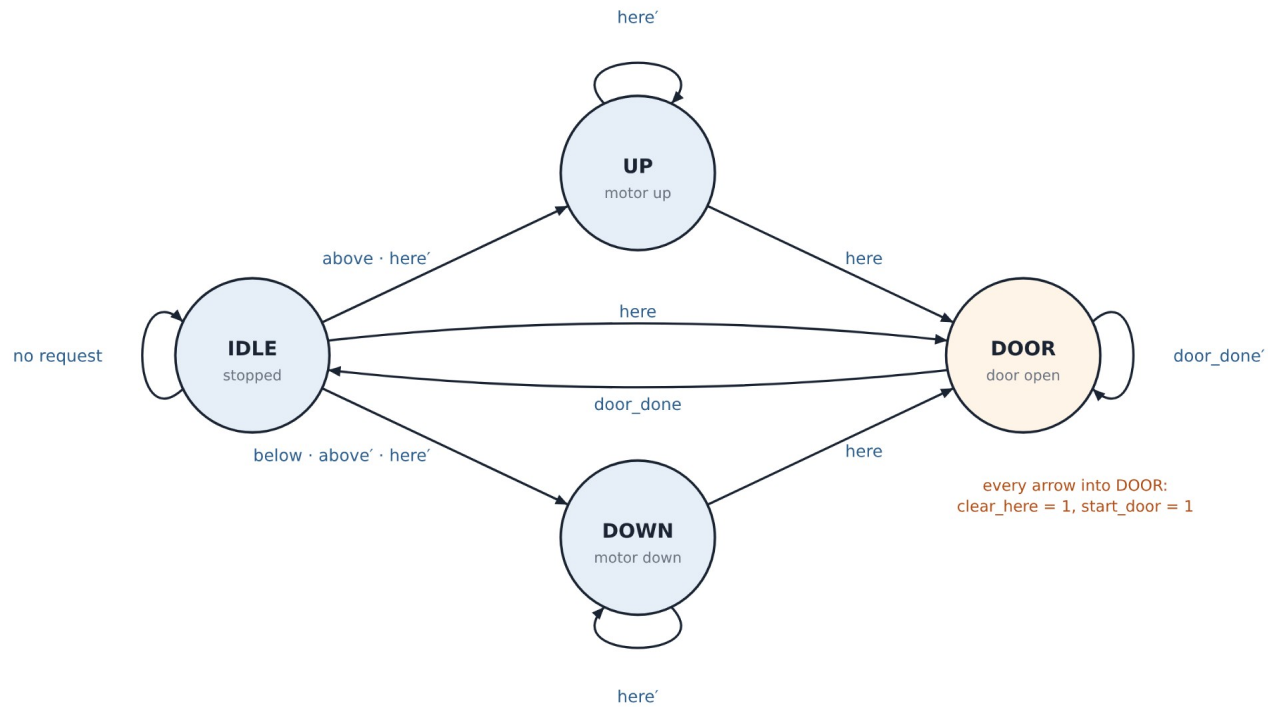
8 floors as one FSM: 2^8 request patterns $\times$ 8 floors $\times$ 4 = **8,192** states. The controller needs 4.

Status: **here, above, below, door_done**.

Control: **up, down, clear_here, start_door**.

MORE EXAMPLES · LIFT

The lift controller



IDLE looks in order: a request **here**, then **above**, then **below**.

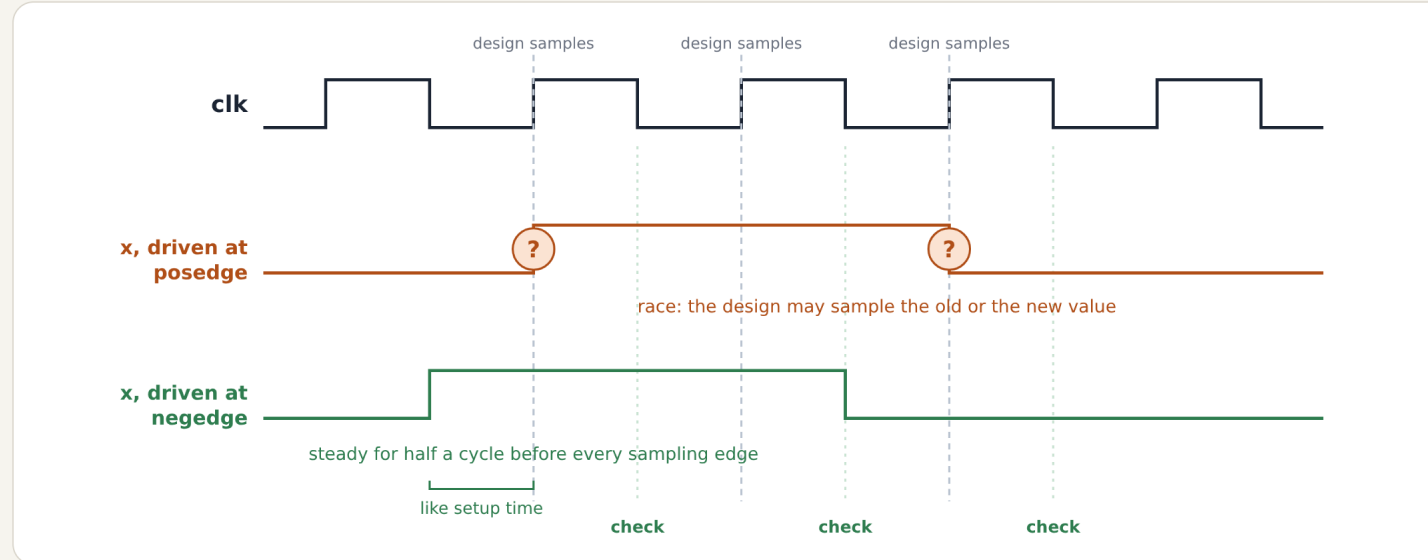
UP and DOWN run the motor until here = 1; the floor register counts each arrive pulse.

Entering DOOR: **clear_here** (request served) and **start_door**.

Four states, whatever the number of floors.

TESTING

When does the testbench act?



The race

Change an input at the rising edge and the design may see the old or the new value. Verilog does not say which.

The rule

Drive inputs at **@(negedge clk)**: steady half a cycle before the design samples them.

Only the testbench

The design never uses negedge. Every flip-flop stays on posedge clk.

TESTING

The negedge pattern in code

Right

```
always #5 clk = ~clk;           // 10 ns clock

// a D flip-flop under test: q <= d
@(negedge clk) d = 1;           // drive between edges
@(negedge clk) check(q, 1'b1);  // after one rising edge
                             d = 0;
@(negedge clk) check(q, 1'b0);
```

Wrong

```
@(posedge clk) d = 1;           // races the flip-flop
```

One @(negedge clk) = one clock cycle. The testbench reads like the cycle trace.

Moore and registered outputs: check at the next negedge, after the rising edge.

Mealy outputs follow the input at once: check a moment after the change, #1, in the same cycle. That is why the 101 testbench has #1.

In Icarus, the wrong version made q take the new value **at the same edge**: a flip-flop with no delay. Hardware cannot do that.

TESTING

A self-checking FSM testbench

```
module tb_det101;
  reg clk = 0, reset = 1, x = 0;
  wire z; integer errors = 0;
  det101_mealy dut (.clk(clk), .reset(reset), .x(x), .z(z));
  always #5 clk = ~clk;
  task send(input b, input exp_z);
    begin
      @(negedge clk) x = b;
      #1 if (z !== exp_z) begin
        errors = errors + 1;
        $display("FAIL x=%b z=%b", x, z);
      end
    end
  endtask
endmodule
```

```
initial begin
  @(negedge clk) reset = 0;
  send(1,0); send(0,0); send(1,1);
  send(0,0); send(1,1); // overlap
  send(1,0); send(0,0); send(0,0);
  send(1,0);
  if (errors == 0) $display("PASS");
  else $display("FAIL: %0d", errors);
  $finish;
end
endmodule
```

TESTING

What a state machine test must cover

Reset

After reset: start state, all outputs at their idle values.

Every arrow

Each transition of the diagram at least once. Count them: the 101 detector has 6.

Overlap and repeats

1 0 1 0 1; long runs; the same input many cycles in a row.

Reset mid-sequence

Assert reset in every state: the machine must return to S0 at once.

One-cycle events

Outputs such as done: high for exactly one cycle, not two.

Unused states

Hard to reach in simulation. Argue from the default branch, or force the state in the testbench.

TESTING THE UNIT

What the testbench checks

All 256 pairs

$\text{sum} = (a + b) \bmod 16$ for $N = 4$.

Latency

done exactly $N + 1$ cycles after start, for one cycle.

Start held high

A second cycle of start, with new inputs, changes nothing.

RULES

Six rules for every FSM in the labs

One clocked block per register group

State register alone; datapath registers together. All with `<=`.

@(*) blocks assign everything

Default values at the top. Vivado latch warnings are errors.

Always a default branch

Unused codes go to the idle state.

Never a derived clock

Every register on `clk`. Slow events use an enable such as a tick.

Events are one cycle

done, load, a pulse: check they last exactly one cycle.

Inputs latched at start

Data the job needs is loaded into the datapath, not read live.

EXERCISE

Your turn: slow the adder down with a tick

A new input **tick** is 1 for one cycle every so often. Change the serial adder so it adds **one bit per tick**, not one bit per clock. Every register stays on clk.

a

Which controller outputs change?
Write the new assign lines.

b

Which arrow of the state diagram changes? Write the new case line.

c

Does the datapath change at all?

ANSWER

Tick as an enable

```
assign shift = (state == ADD) && tick;  
  
ADD: next = (last && tick) ? DONE : ADD;
```

a

shift becomes a Mealy output:
shift only on tick cycles. busy, load,
done unchanged.

b

Leave ADD only on the tick that does
the last shift.

c

No change. The datapath only obeys
shift; it never sees the clock rate.

Simulated with a random tick: all 256 sums correct.

SUMMARY

The diagram is the code

Three blocks

State register, next-state case, output assigns.

Case = diagram

One case item per state, one line per arrow, and a default.

Test the arrows

Self-checking: reset, every transition, one-cycle events.

Controller + datapath

The FSM decides, the datapath does. Counters keep the FSM small.

In the lab: the serial transmitter and receiver are this pattern: shift register + counter + FSM.