```
module i2c_master (
    input logic clk, reset,
```

→ could have used wire, but it's preferred SystemVerilog standard for signals.
we can run continous (assign) or procedural assignment.

```
    [7:0] din,
```

→ Writing: 8-bit data to slave
Reading: din[0] for acknow — that we have read.

```
    [15:0] dvsr,
    [2:0] cmd,
```

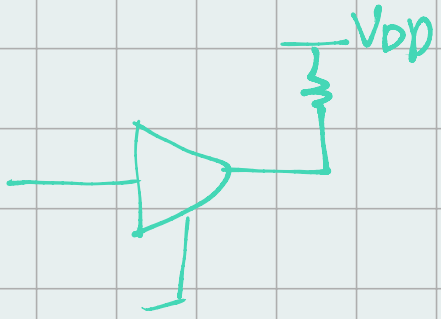
→ to tell FSM what operation to perform (start stop — rw—)
5 states in total)

```
    wr_i2c,
```

→ when cmd is received, then wr_i2c goes high for one cycle.
Tells FSM to execute that command.

```
    tri scl,
    tri sda,
```

inout → bidirectional
→ tri state buffer



when 0 → line 1'b0
1 → line 1'bz (open)

letting our external pull-up resistor to pull line high.

---

ready, done_tick, ack,

when 1, module is in idle or hold state.

goes high for a moment a byte (read or write) is completed.
- used to know when dout is valid

0 → slave acknowledge the byte.
1 → slave did not acknowledge or is busy.

[7:0] dout);

localparam

```
Start   = 0 ;
Write   = 1 ;
read.   = 2 ;
Stop    = 3 ;
Restart = 4 ;
```

typedef enum {...} state_type;

state_type _____ state_reg, state_next;

_____ c_reg, c_next → counter

_____ quar, half → 2 clk pulses

↳ 1 clk pulses

— [8:0] tx_reg, tx_next

↳ 8 for data, 1 for ack

— [8:0] rx, reg, rx, next.

_____ cmd reg _____

— [3:0] bit reg → to count the 9 bits

— sda_out, scl_out, sda_reg, scl_reg

data_phase.

↳ 0 or 1 to determine if master wants to drive or not

↳ when in data phase.

— into → when master should be listening.

→ treat sda as input.

— nack → din [0]

// output control logic

① always @ (_____)

if (reset)

_____ = 1'b1; ↗ the idle state in I2C is pulled high

else

$\_reg = \_out;$

② `assign scl = (scl_reg) ? 1'bz : 1'b0`

open-drain (0, z)
not tri-state buffer (0, 1, z)

↳ Here we used open drain logic because it allows slave to stretch the clock.

ie. • If master want scl = 1, scl_reg = 1. driver goes z, then pull up resistor makes the line 1.

• If slave is not ready, it can pull the line low (0).

• If used tri-state driver, the master would actively drive '1' & the slave pulling '0' would cause a short circuit.

③ `assign bit0 = (——— || ———)`

↳ to declare when master should release the sda line to listen.

↳ (data_phase && cmd_reg == RD_cmd && bit_reg < 8) : Reading data.

↳ during a read command, for the first 8 bits

↳ (data_phase && cmd_reg == WR_cmd && bit_reg == 8) ; Receiving ack.

↳ during write command, on the 9th bit, the master should listen for the slave ack.

④ assign sda = (into || sda_reg) ? 1'bz : 1'b0;

when listening release the line.

we wants send a line

⑤ —— dout = rx_reg[8:1];
ack = rx_reg[0];
nack = din[0];

// fsmd (with datapath)

① always @ (——)
→ FSM to idle state & clear all counts.
→ every clk pulse.

state ++	cmd_reg <= next
c_reg ++	tx ——
bit_reg ++	rx ——

② assign qutr = dvsr;
assign half = (qutr * 2) to get half period

next-state Logic

case (state_reg)

idle: ready_i = 1'b1;
if (wr_i2c && cmd == start)
next_state

start1: sda_out = 1'b0,
waits for half a period.
next_state.

start2: scl_out = 1'b0
waits for quarter wave
next_state.

hold: ready_i = 1'b1.
scl_out = 1'b0
sda_out = 1'b0
waits for wr_i2c
when it gets a command.
stop_CMD → goes to stop1.
Restart CMD → goes to restart.
wr CMD or rd_CMD

reset bit counter.
Load the transmit register
next state = data1.

data1: scl_out = 1'b0
sda_out = tx_reg[0]
wait qutr.
then to data2.

data2: scl_out = 1'b1.
rx_next = {rx_reg[7:0], sda};
wait qutr.
next state

data3: scl_out = 1'b1.
wait qutr
next state

data4: scl_out = 1'b0;
wait qutr.
if (bit_reg == 8).
↳ ✓ done_tick = 1 & go to data end
↳ × bit_next ++, shift tx_reg. left & goes to data2.

data_end: to keep lines low &
wait qutr
goes to hold to wait for
next command.

restart: wait half
c_out <= 0
goes to start 1.

stop 1: sda_out = 1'b0.
wait half.
goes to stop 2.