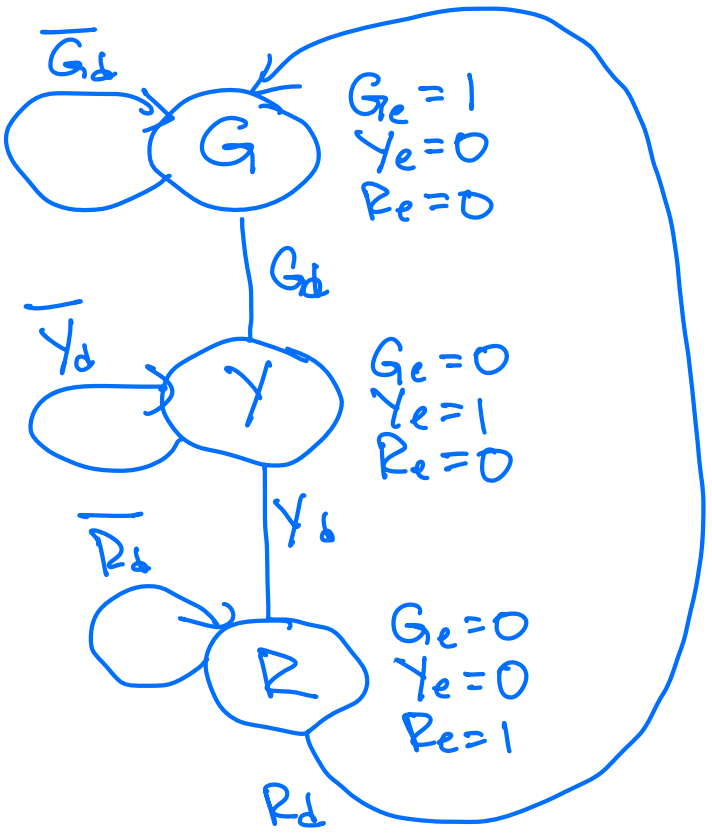
Traffic controller



circuit will have 2 input (clock, enable)
3 outputs (R, G, B lines for lights)

```
module TRAFFIC(
    input clk, ena,
    output reg R, G, Y );
```

no need for wire here, it's the default

can put type declaration here

need 3 timers. Lets assume clk has f = 1kHz
so period T = 1ms

want G, R lights on for 30s
Y " " " 2s

for G, R, 30s at 1ms/tick = 30k ticks

need register that can count to 30k

for 15 bits, $2^{15} = 32768$

for Y, 20s ⇒ 2k ticks so need 11 bits: $2^{11} = 2048$

decimal 30,000 = Hex 753Ø ⇒ 15'h753Ø

decimal 2,000 = Hex 7DØ ⇒ 11'h7DØ

(11 = #bits; 'h ⇒ hex; 'd ⇒ decimal)

Counters:

```
reg [14:0] rT, gT;
reg [10:0] yT;
```

then a done line for each timer

```
wire rD, gD, yD;

assign rD = (rT == 'd30000);
assign gD = (gT == 'd30000);
assign yD = (yT == 'd2000);
```

parenthese optional

counters

```
always @(posedge clk) begin
   if (R) rT <= rT + 1;
   else   rT <= 0;      (don't need 'd00000)
   if (G) gT <= gT + 1;
```

```
    else gT <= 0;
    if (Y) yT <= yT+1;
    else yT <= 0;
end
```

next comes state machine

=> 3 states RED, GREEN, YELLOW

```
parameter [1:0] RED=0, GREEN=1, YELLOW=2;
```

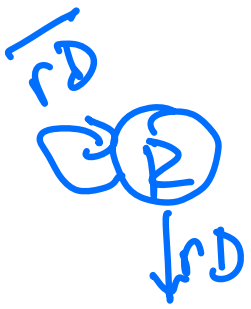
2 bits to hold numbers 0, 1, 2

FSM has 3 states so need 2-bit state bus

```
reg [1:0] state;

always @(posedge clk) begin
    if (ena) begin
        case (state)
            RED: begin
                R <= 1;
                G <= 0;
                Y <= 0;
                if (rD) state <= GREEN;
                else state <= state;
            end
```

} specify every output reg!



```
GREEN: begin
    R<=0;
    G<=1;
    Y<=0;
    if (gD) state<=YELLOW;
    else state<=GREEN
end
YELLOW: begin
    R<=0;
    G<=0;
    Y<=1;
    if (yD) state<=RED;
    else state<=YELLOW;
end
end
end
else begin
    R<=1;
    G<=0;
    Y<=0;
    state<=RED;
endcase
end
```

← ends if (end) begin

← turns on red light

note when end is asserted, red counter has been counting so will go to red state for unknown time, prob of

← ends case switching

← ends always @ ... begin

Reset

above we used cnd signal
we could also use a reset signal that puts FSM into determined state

2 types of reset: synchronous } with respect
asynchronous } to clock

synchronous

```
always @(posedge clk) begin
    if (reset) begin
        state <= RED;
        R <= 1;
        G <= 0;
        Y <= 0;
    end
    else begin
        case (state)
            :
        end case
    end
```

← only @ posedge clk so will effect FSM synchronous w/clk

asynchronous

```
always @(posedge clk or posedge reset)
    ⋮
```

not synch'd w/ clock!