look-ahead adder, the carry signals are calculated in advance, based on the input signals. For any bit position i, one can see that a carry will be generated if the

corresponding input bits (i.e., A_i, B_i) are 1 or if there was a carry-in to that bit and at least one of the input bits are 1. In other words, bit i has carry-out if A_i and B_i are 1 (irrespective of carry-in to bit i); bit i also has a carry-out if $C_i = 1$ and either A_i or B_i is 1. Thus, for any stage i, the carry-out is

$$C_{i+1} = A_iB_i + (A_i \oplus B_i) \cdot C_i \tag{4-1}$$

The "$\oplus$" stands for the exclusive OR operation. Equation 4-1 simply expresses that there is a carry out from a bit position if it **generated** a carry by itself (e.g., $A_iB_i = 1$) or it simply **propagated** the carry from the lower bit forwarded to it (i.e., $(A_i \oplus B_i) \cdot C_i$).

Since $A_iB_i = 1$ indicates that a stage generated a carry, a general **generate (Gi)**

$$S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i \tag{4-5}$$

The expression $P_i \oplus C_i$ can be used for sum only if P_i is defined as $A_i \oplus B_i$.

The carry-out equation can be rewritten by substituting (4-2) and (4-3) in (4-1) for G_i and P_i as:

$$C_{i+1} = G_i + P_i C_i \tag{4-6}$$

In a 4-bit adder, the $C_i s$ can be generated by repeatedly applying Equation 4-6 as shown here

$$C_1 = G_0 + P_0 C_0 \tag{4-7}$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0 \tag{4-8}$$

$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0 \tag{4-9}$$

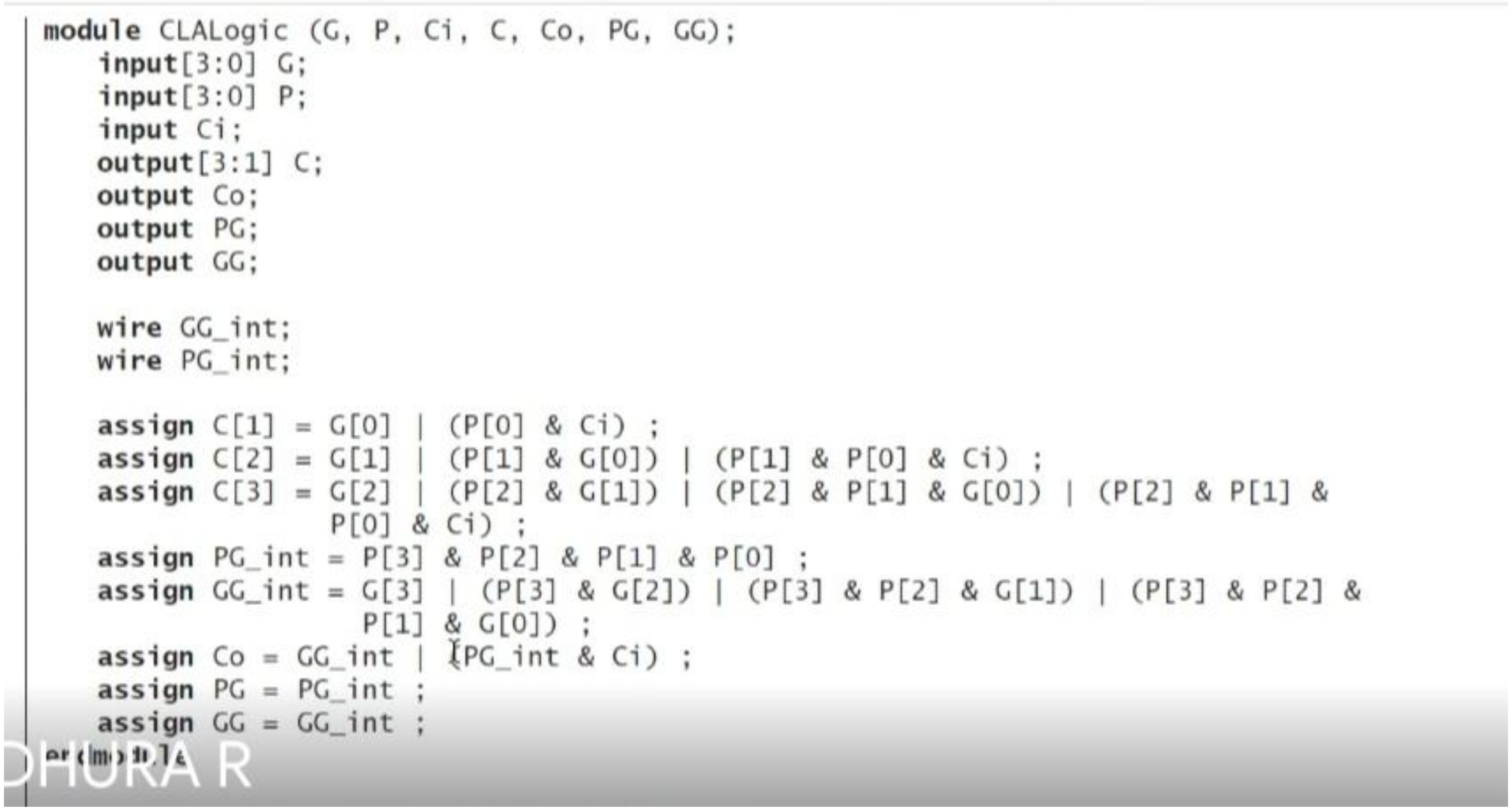
$$C_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0 \tag{4-10}$$

These carry bits are the look-ahead carry bits. They are expressed in terms of $P_i s$, $G_i s$, and C_0. Thus, the sum and carry from any stage can be calculated without

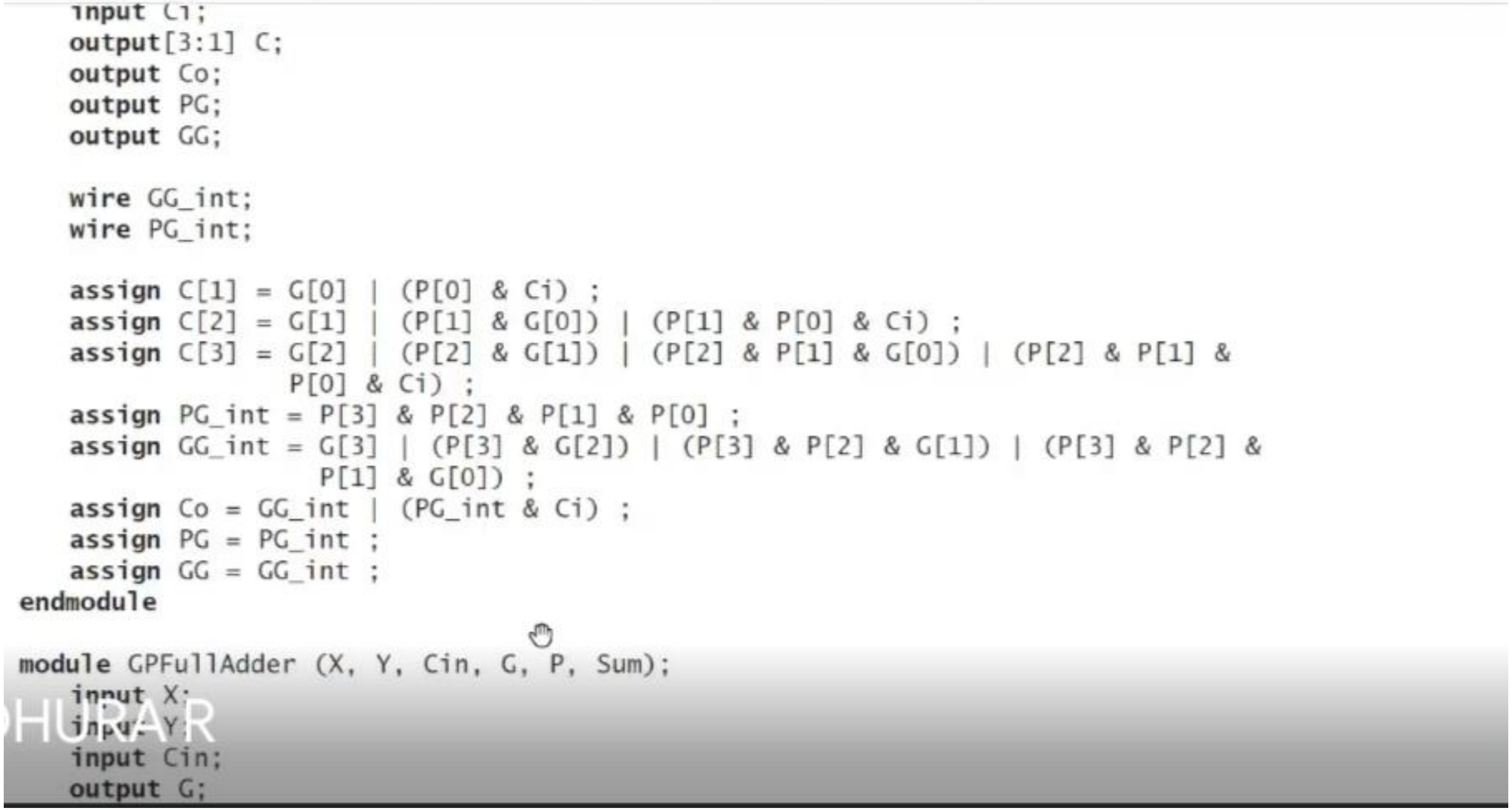
Figure 4-10 illustrates the Verilog description of a 4-bit carry look-ahead ad

FIGURE 4-10: Verilog Description of a 4-Bit Carry Look-Ahead Adder

```
module CLA4 (A, B, Ci, S, Co, PG, GG);
   input[3:0] A;
   input[3:0] B;
   input Ci;
   output[3:0] S;
   output Co;
   output PG;
   output GG;

   wire[3:0] G;
   wire[3:0] P;
   wire[3:1] C;
   CLALogic CarryLogic (G, P, Ci, C, Co, PG, GG);
   GPFullAdder FA0 (A[0], B[0], Ci, G[0], P[0], S[0]);
   GPFullAdder FA1 (A[1], B[1], C[1], G[1], P[1], S[1]);
   GPFullAdder FA2 (A[2], B[2], C[2], G[2], P[2], S[2]);
   GPFullAdder FA3 (A[3], B[3], C[3], G[3], P[3], S[3]);
endmodule
```



```
module CLALogic (G, P, Ci, C, Co, PG, GG);
    input[3:0] G;
    input[3:0] P;
    input Ci;
    output[3:1] C;
    output Co;
    output PG;
    output GG;

    wire GG_int;
    wire PG_int;

    assign C[1] = G[0] | (P[0] & Ci) ;
    assign C[2] = G[1] | (P[1] & G[0]) | (P[1] & P[0] & Ci) ;
    assign C[3] = G[2] | (P[2] & G[1]) | (P[2] & P[1] & G[0]) | (P[2] & P[1] &
                  P[0] & Ci) ;
    assign PG_int = P[3] & P[2] & P[1] & P[0] ;
    assign GG_int = G[3] | (P[3] & G[2]) | (P[3] & P[2] & G[1]) | (P[3] & P[2] &
                    P[1] & G[0]) ;
    assign Co = GG_int | (PG_int & Ci) ;
    assign PG = PG_int ;
    assign GG = GG_int ;
endmodule
```



```
    input Ci;
    output[3:1] C;
    output Co;
    output PG;
    output GG;

    wire GG_int;
    wire PG_int;

    assign C[1] = G[0] | (P[0] & Ci) ;
    assign C[2] = G[1] | (P[1] & G[0]) | (P[1] & P[0] & Ci) ;
    assign C[3] = G[2] | (P[2] & G[1]) | (P[2] & P[1] & G[0]) | (P[2] & P[1] &
                  P[0] & Ci) ;
    assign PG_int = P[3] & P[2] & P[1] & P[0] ;
    assign GG_int = G[3] | (P[3] & G[2]) | (P[3] & P[2] & G[1]) | (P[3] & P[2] &
                    P[1] & G[0]) ;
    assign Co = GG_int | (PG_int & Ci) ;
    assign PG = PG_int ;
    assign GG = GG_int ;
endmodule

module GPFullAdder (X, Y, Cin, G, P, Sum);
    input X;
    input Y;
    input Cin;
    output G;
    assign Sum33 = A + B + Ci ;
    assign S = Sum33[31:0] ;
    assign Co = Sum33[32] ;
endmodule
```

Example

If gate delays are t_g, what is the delay of the fastest 32-bit adder? Assume that the amount of hardware consumed is not a constraint. Only speed is important.

Answer: One can express each sum bit of a 32-bit adder as a sum-of-products expression of the input bits. There will be 33 such equations, including one for the carry out bit. These equations will be very long, and some of them could include 60+ variables in the product term. Nevertheless, if gates with any number of inputs are available, theoretically a 2-level adder can be made. Although it is not very practical, theoretically, the delay of the fastest adder will be $2t_g$ if gate delays are t_g.

Example

Is ripple-carry adder the smallest 32-bit adder?

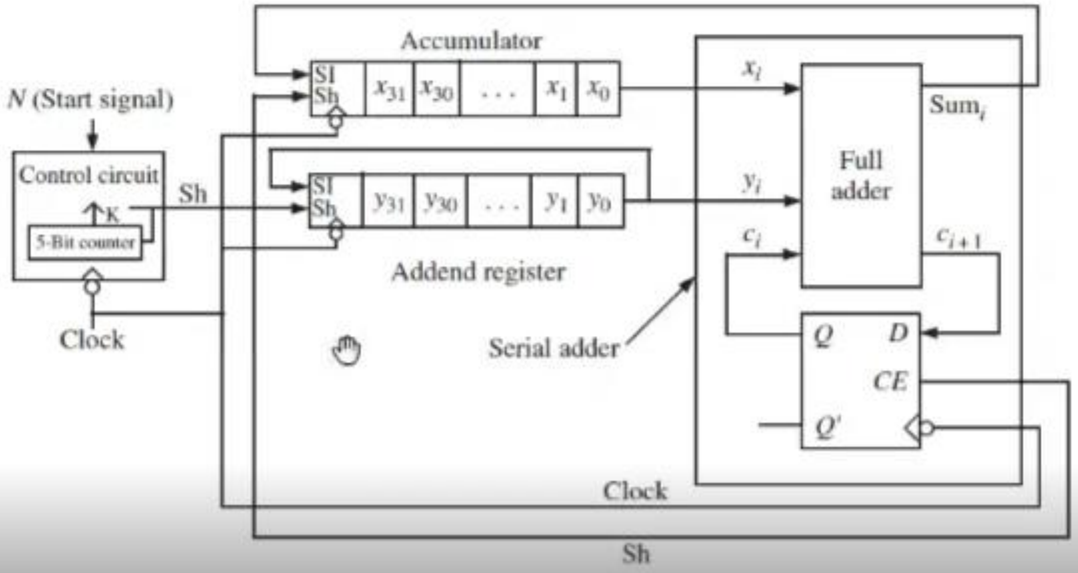
Answer: A 32-bit ripple-carry adder uses 32 1-bit adders. One could design a 32-bit serial adder using a single 1-bit full adder. The input numbers are shifted into the adder, one bit at a time, and carry output from addition of each pair of bits is saved in a flip-flop and fed back to the next addition. The hardware illustrated in Figure 4-12 accomplishes this. The delay

Example

Is ripple-carry adder the smallest 32-bit adder?

Answer: A 32-bit ripple-carry adder uses 32 1-bit adders. One could design a 32-bit serial adder using a single 1-bit full adder. The input numbers are shifted into the adder, one bit at a time, and carry output from addition of each pair of bits is saved in a flip-flop and fed back to the next addition. The hardware illustrated in Figure 4-12 accomplishes this. The delay

FIGURE 4-12: A 32-Bit Serial Adder Built from a Single 1-Bit Adder



nents available in the target technology. For instance, if you are using an FPGA with fast adder support, the software may map some of the functions into the fast adder circuitry. Depending on the number of FPGA logic blocks and interconnects used, the delays will be different from the manual calculations. The delays of a ripple-carry, carry look-ahead, and serial adder for a gate-based implementation are presented in Table 4-1 for various adder sizes. One can see that the carry look-ahead adder is very attractive for large adders.

BLE 4-1: Comparison ipple-Carry and Carry Look-Ahead Adders

Adder size	Ripple-carry adder delay	CLA delay	Serial adder delay
4 bit	8 t_g	5-6 t_g	16 t_g
16 bit	32 t_g	7-8 t_g	64 t_g
32 bit	64 t_g	9-10 t_g	128 t_g
64 bit	128 t_g	9-10 t_g	256 t_g

4.4 Traffic Light Controller

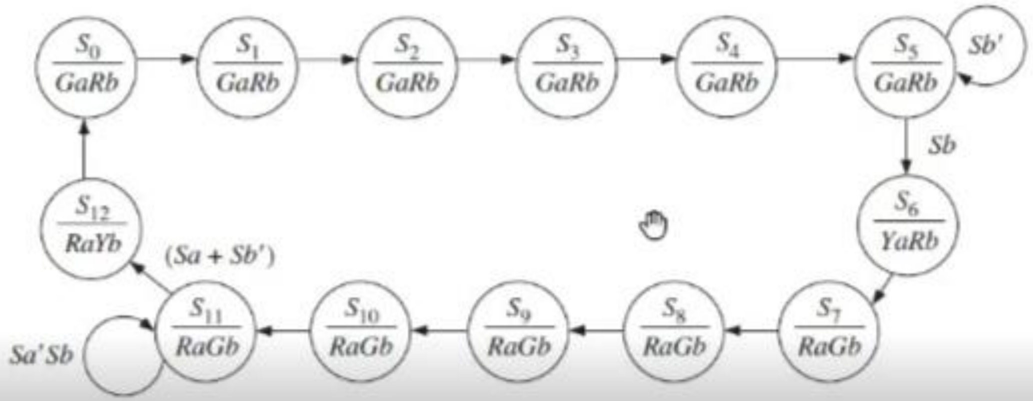
4.4 Traffic Light Controller

Let us design a sequential traffic light controller for the intersection of street "A" and street "B." Each street has traffic sensors, which detect the presence of vehicles approaching or stopped at the intersection. $Sa = 1$ means a vehicle is approaching on street "A," and $Sb = 1$ means a vehicle is approaching on street "B." Street "A" is a main street and has a green light until a car approaches on "B." Then the lights change, and "B" has a green light. At the end of 50 seconds, the lights change back unless there is a car on street "B" and none on "A," in which case the "B" cycle is extended for 10 additional seconds. If cars continue to arrive on street "B" and no car appears on street "A," "B" continues to have a green light. When "A" is green, it remains green at least 60 seconds, and then the lights change only when a car approaches on "B." Figure 4-13 shows the external connections to the controller. Three of the outputs (*Ga*, *Ya*, and *Ra*) drive the green, yellow, and red lights on

RA R

Figure 4-14 shows a Moore state graph for the controller. For timing purposes, the sequential circuit is driven by a clock with a 10-second period. Thus, a state change can occur at most once every 10 seconds. The following notation is used: *GaRb* in a state means that $Ga = Rb = 1$ and all the other output variables are 0. $Sa'Sb$ on an arc implies that $Sa = 0$ and $Sb = 1$ will cause a transition along that arc. An arc without a label implies that a state transition will occur when the clock occurs, independent of the input variables. Thus, the green "A" light will stay on for six clock cycles (60 seconds) and then change to yellow if a car is waiting on street "B."

: State
affic Light



R

The Verilog code for the traffic light controller (Figure 4-15) represents the state

FIGURE 4-15: Verilog Code for Traffic-Light Controller

```
module traffic_light (clk, Sa, Sb, Ra, Rb, Ga, Gb, Ya, Yb);
input clk;
input Sa;
input Sb;
inout Ra;
inout Rb;
inout Ga;
inout Gb;
inout Ya;
inout Yb;

reg Ra_tmp;
reg Rb_tmp;
reg Ga_tmp;
reg Gb_tmp;
reg Ya_tmp;
reg Yb_tmp;

reg[3:0] state;
reg[3:0] nextstate;
```

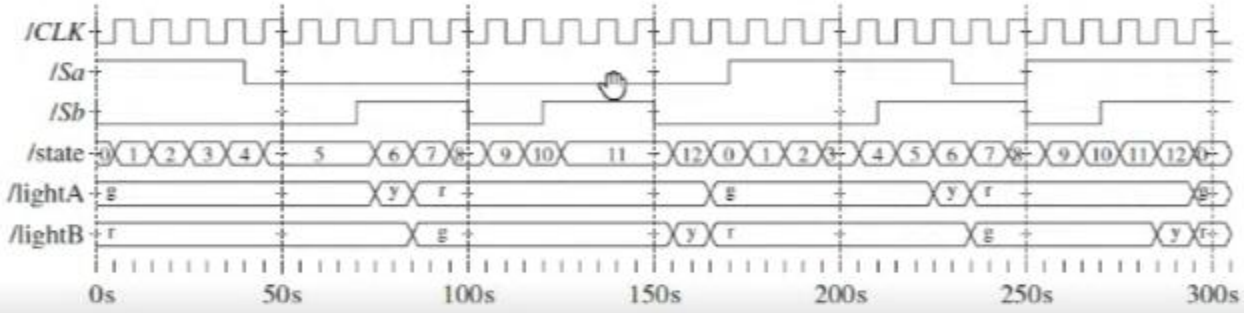
```
always @(posedge clk)
begin
   state <= nextstate ;
end

assign lightA = (Ra==1'b1) ? R : (Ya==1'b1) ? Y : (Ga==1'b1) ? G : lightA;
assign lightB = (Rb==1'b1) ? R : (Yb==1'b1) ? Y : (Gb==1'b1) ? G : lightB;

endmodule
```

For each state, only the signals that are 1 are listed within the case statement. Since in Verilog a signal will hold its value until it is changed, we should turn off each signal when the next state is reached. In state 6 we should set *Ga* to 0, in state 7 we should set *Ya* to 0, and so forth. This could be accomplished by inserting appropriate statements in each case. For example, we could insert `Ga <= '0'` in the **`case of state=6 -> case 6.`** An easier way to turn off the outputs is to set them all to 0 before the case statement, as shown in Figure 4-15. At first, it seems that a glitch might occur in the output when we set a signal to 0 that should remain 1. However, this is not a problem, because the sequential statements within an always state-the light is yellow, and to G when the light is green. The following simulator command file first tests the case where both self-loops on the graph are traversed and then the case where neither self-loop is traversed:

```
add wave clk Sa Sb state lightA lightB
force clk 0 0, 1 5 sec -repeat 10 sec
force Sa 1 0, 0 40, 1 170, 0 230, 1 250 sec
force Sb 0 0, 1 70, 0 100, 1 120, 0 150, 1 210, 0 250, 1 270 sec
run 300 sec
```

The test results in Figure 4-16 verify that the traffic lights change at the specified times.

t
: Light


/CLK
/Sa
/Sb
/state
/lightA
/lightB
0s
50s
100s
150s
200s
250s
300s