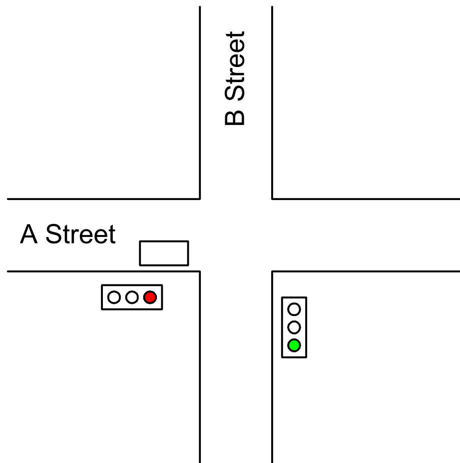
CS99S
Laboratory 6 Preparation

Copyright © W. J. Dally 2001
November 5, 2001

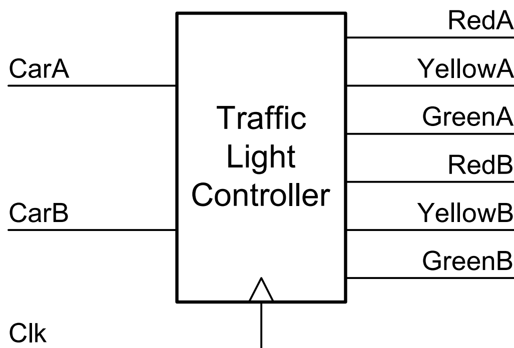
Objectives:

1. Understand principles of *one-hot* finite state machine design
2. Build a simple traffic light controller state machine

A Traffic Light Controller



Consider the problem of controlling a traffic light at the intersection of two equally busy streets, A Street and B Street. Our traffic light controller takes two inputs – CarA (which is high when there is a car just before the intersection on A Street – in either direction), and CarB (which is high when there is a car just before the intersection on B street). The controller needs to generate six outputs – RedA, YellowA, GreenA, RedB, YellowB, and GreenB – which drive the respective traffic lights for A Street and B Street. In the figure above, CarA will be high, since there is a car (the rectangle) on A Street, and CarB will be low, since there is no car on B Street. Also in the Figure RedA is high since A Street has a red light, and GreenB is high since B Street has a green light. All other outputs are low. We can think of the traffic light controller as a black box that takes two inputs (and a clock) and generates six outputs as shown below. Our job is to design the logic inside the box.

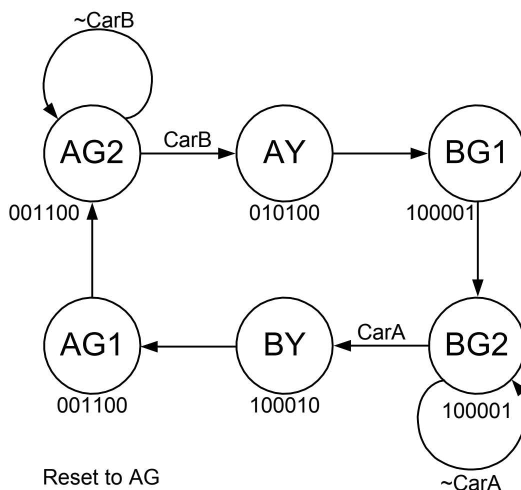


Our design must obey the following rules:

1. When the light is green on A Street and a car is waiting on B Street, give A Street a yellow light for one clock cycle and then give A Street a red light and B Street a green light for at least two cycles.
2. When the light is green on A Street and there is no car on B Street, leave the light green on A Street.
3. When the is green on B Street (and we've finished the two cycles from step 1) and a car is waiting on A Street, give B Street a yellow light for one clock cycle and then give B Street a red light and A Street a green light for at least two cycles.
4. When the light is green on B Street and there is no car on A Street, leave the light green on B Street.
5. When you press the reset switch, after no more than six cycles, the light should be initially green on A Street and red on B Street and the controller should be ready for operation.

State Diagram

We can translate these five rules into the following state diagram. For clarity, we omit the transitions that take all states to state AG2 (A Green 2nd cycle) when reset is true.



Each circle in the state diagram represents a state. The name of the state is in the circle and the state of the six output lines (in the order listed above) is shown below that state. The transitions between the states are labeled with the signals that make these transitions occur. Most of the edges have no label which indicates that the transition always occurs (unless reset is asserted).

When our finite-state machine (FSM) is in state AG2, A Street has a green light and B Street has a red light. The transition from AG2 back to itself indicates that as long as there is no car on B Street we keep the A Street light green. The transition to AY (for A Yellow) indicates that if there is a car on B Street, we make the A Street light yellow on

the next cycle. AY always transitions to BG1 (for B Green 1st cycle) where the A Street light becomes red and the B Street light becomes green. BG1 always transitions to BG2 where the FSM waits for a car on A Street before sequencing through BY and AG1 back to AG2.

From this state diagram we can write the following state table:

	Inputs				Outputs	
State	reset	CarA	CarB	nextState	A lights	B lights
*	1	*	*	AG2	Green	Red
AG2	0	*	0	AG2	Green	Red
AG2	0	*	1	AY	Green	Red
AY	0	*	*	BG1	Yellow	Red
BG1	0	*	*	BG2	Red	Green
BG2	0	0	*	BG2	Red	Green
BG2	0	1	*	BY	Red	Green
BY	0	*	*	AG1	Red	Yellow
AG1	0	*	*	AG2	Green	Red

Prep Question 1: For your lab you are going to build a slightly different traffic light controller that is designed for the case where A Street is a busy street and B Street is a quiet street. Your controller will obey the following rules:

1. When the light is green on A Street and a car is waiting on B Street , give A Street a yellow light for one clock cycle and then give A Street a red light and B Street a green light for at least **one** cycle.
2. When the light is green on A Street and there is no car on B Street, leave the light green on A Street.
3. When the light is green on B Street (after the one cycle from step 1) and there is either no car on B Street **or** a car on A Street, give B Street a yellow light for one clock cycle and then give B Street a red light and A Street a green light for at least **three** cycles.
4. When the light is green on B Street and there is no car on A Street **and** there is a car on B street, leave the light green on B Street.
5. There is no need for a reset switch. Your FSM will be initialized by holding carA and carB both low for enough cycles for the FSM to 'home' to a known state.

Write a state diagram and a state table for your traffic light controller.

One-Hot State Encoding

The traffic light controller above (not the one from your prep question) has six states: AG2, AY, BG1, BG2, BY, AG1. To design logic for the machine we need to assign bit patterns (binary numbers) to these states. The mapping between states and bit patterns is called a *state assignment* or *state encoding*.

State	One-Hot	Min Size
AG2	100000	000
AY	010000	001
BG1	001000	010
BG2	000100	110
BY	000010	101
AG1	000001	100

Two possible state encodings (of many possible ones) are shown in the table above. The encoding on the left is called a *one-hot* encoding since each state is represented by a state vector with exactly one bit set. Since there are six states, this representation requires six bits. Each bit corresponds to a particular state. When the bit is set, the FSM is in that state. Only one bit can be set at a time – hence the name one hot. While it requires more state bits, a one-hot encoding often simplifies the logic of the state machine.

The other encoding is a minimum-size encoding. To represent six states we need at least three bits. This encoding represents one way of mapping the six states onto three-bit binary numbers. To simplify the logic the encodings were chosen with a minimum number of 1s and with similar states having encodings that differ in as few bit positions as possible.

The next-state logic for a one-hot FSM (an FSM using a one-hot encoding) can be generated directly from the state diagram. There is a logic equation for each state. This equation is the OR of a number of AND terms, one for each arrow into that state. Each of the AND terms ANDs the state at the tail of the arrow with the input conditions labeling the arrow. For example, ignoring reset, the equation for state variable AG2 is

$$AG2 = AG1 \vee (AG2 \wedge \sim CarB)$$

There are two AND terms, one (AG1) for the arrow from AG1 to AG2, and the second ($AG2 \wedge \sim CarB$) for the arrow from AG2 to itself. Following a similar methodology we can write the equations for all of six the state variables (again ignoring reset):

$$nextAG2 = AG1 \vee (AG2 \wedge \sim CarB)$$
$$nextAY = AG2 \wedge CarB$$
$$nextBT = AY$$
$$nextBG2 = BG1 \vee (BG2 \wedge \sim CarA)$$
$$nextBY = BG2 \wedge CarA$$
$$nextAT = BY$$

Prep Question 2: Ignoring reset, write the logic equations for the next-state variables of your traffic light controller from Prep Question 1 using a one-hot state assignment.

Homing Sequences

Now lets deal with the issue of reset. We could ensure that the machine transitions directly to state AG2 on the first clock cycle after the reset button is pushed by inserting five edges into the graph. The logic equations then (after some simplification for AG2) become:

$$nextAG2 = AG1 \vee (AG2 \wedge \sim CarB) \vee reset$$
$$nextAY = AG2 \wedge CarB \wedge \sim reset$$
$$nextBT = AY \wedge \sim reset$$
$$nextBG2 = (BG1 \vee (BG2 \wedge \sim CarA)) \wedge \sim reset$$
$$nextBY = BG2 \wedge CarA \wedge \sim reset$$
$$nextAT = BY \wedge \sim reset$$

This complicates the logic equations a great deal, since every state needs to include a ~reset – think of all of those extra gates you would have to wire up to do this. If we insist that CarB be false during reset we can get by with only resetting states AG2 and BG2 as follows:

$$nextAG2 = AG1 \vee (AG2 \wedge \sim CarB) \vee reset$$
$$nextAY = AG2 \wedge CarB$$
$$nextBT = AY$$
$$nextBG = (BG1 \vee (BG2 \wedge \sim CarA)) \wedge \sim reset$$
$$nextBY = BG2 \wedge CarA$$
$$nextAT = BY$$

This works because state bits AY and BG2 are guaranteed to be zero after one cycle. This makes state bits BG1 and BY zero on the second cycle. Finally, state bit BY is cleared on the third cycle and the machine is in reset.

A sequence like this, that takes the machine to a known state (usually the reset state) from any initial state, is called a *homing sequence*. In practice such sequences are rarely used. Its usually better to burn a few more gates and get the machine reset in one cycle. However, since we're wiring this state machine by hand, its worth playing a few tricks to save some gates and wires.

Prep Question 3: Ensure that your FSM homes to a known state when carA and carB are both held low. This should already be the case.

Output Variables

Now that our next state logic is complete we need to worry about how to generate the six outputs from our machine. This can also be done by examining either the state diagram or the state table. The equation for each output variable is the OR of those state

variables during which the output is true. For example, since RedA is true in states BG1, BG2, and BY the equation for RedA is:

$$RedA = BG1 \vee BG2 \vee BY$$

We can write the equations for all six of the outputs in a similar manner.

$$RedA = BG1 \vee BG2 \vee BY = BY \vee GreenB$$
$$YellowA = AY$$
$$GreenA = AG1 \vee AG2$$
$$RedB = AG1 \vee AG2 \vee AY = AY \vee GreenA$$
$$YellowB = BY$$
$$GreenB = BG1 \vee BG2$$

Upon review by the company's lawyers it is pointed out that during the reset homing sequence the lights could momentarily go green in both directions. Since this could result in a horrible accident, the lawyers insist that you make both lights go red during the reset sequence:

$$RedA = BY \vee GreenB \vee reset$$
$$YellowA = AY \wedge \sim reset$$
$$GreenA = (AG1 \vee AG2) \wedge \sim reset$$
$$RedB = AY \vee GreenA \vee reset$$
$$YellowB = BY \wedge \sim reset$$
$$GreenB = (BG1 \vee BG2) \wedge \sim reset$$

Prep Question 4: Write the output logic equations for your FSM from Prep Questions 1 through 3. You need not worry about the state of the outputs during the reset sequence.

The Lab

For the lab you will build your traffic light controller and try it out using a DIP switch to generate the CarA and CarB inputs and either your pushbutton or your oscillator to step the clock.

Prep Question 5: In preparation you should draw a complete schematic of your traffic light controller. Your schematic should take two inputs, a clock input, and generate six outputs. Use the 74AC377 octal D-flip-flop to hold your state variables. You can use any of the gate parts 74AC00, AC02, AC08, AC32, AC10, AC20, etc… to implement your next state and output logic.

Prep Question 6 (Optional Challenge Question): Redesign the next state logic and output logic of your traffic light controller using a minimum-size state assignment.

Verilog (Optional)

In practice we like to make sure that our FSM designs work before we go to the trouble of building them. This is particularly true if building the FSM involves making an ASIC which can cost $500,000 and take 3 months.

A popular method of verifying a design is to simulate it using a register-transfer simulator like Verilog. As an example, here is the traffic light controller from above described in Verilog:

```
// tlc.v
// traffic light controller in verilog
// W. J. Dally 10/31/2001
//
module TLC(clk, reset, carA, carB, lightsA, lightsB) ;
  input clk ;                   // clock
  input reset ;                 // reset
  input carA ;                  // a car is waiting on A Street
  input carB ;                  // a car is waiting on B Street

  output[2:0] lightsA ;         // Red, Yellow, Green lights for A Street
  output[2:0] lightsB ;         // Red, Yellow, Green lights for B Street

  reg ag2, ay, ag1, bg2, by, bg1 ; // state bits
  wire nag2, nay, nag1, nbg2, nby, nbg1 ; // next state bits

  wire[5:0] state ; // for observation only

  assign state = {ag2, ay, ag1, bg2, by, bg1} ;

  // state equations
  assign
        nag2 = ag1|(ag2 & ~carB)|reset ,
        nay = ag2 & carB ,
        nbg1 = ay ,
        nbg2 = (bg1|(bg2 & ~carA))&~reset,
        nby = bg2 & carA ,
        nag1 = by ;

  // flip flops
  always @(posedge clk)
    {ag2, ay, ag1, bg2, by, bg1} = {nag2, nay, nag1, nbg2, nby, nbg1} ;

  // output equations
  assign
        lightsA[2] = by | lightsB[0] | reset ,  // red
        lightsA[1] = ay & ~reset ,              // yellow
        lightsA[0] = (ag1 | ag2) & ~reset,      // green
        lightsB[2] = ay | lightsA[0] | reset,   // red
        lightsB[1] = by & ~reset,               // yellow
        lightsB[0] = (bg1 | bg2) & ~reset ;     // green
endmodule
```

And here is the result of simulating it for a few cycles.

```
reset = 1 carA = 0 carB = 0 : lightsA = 100 lightsB = 100 state =xxxxxx
reset = 1 carA = 0 carB = 0 : lightsA = 100 lightsB = 100 state =10x00x
reset = 1 carA = 0 carB = 0 : lightsA = 100 lightsB = 100 state =100000
reset = 1 carA = 0 carB = 0 : lightsA = 100 lightsB = 100 state =100000
reset = 0 carA = 0 carB = 0 : lightsA = 001 lightsB = 100 state =100000
reset = 0 carA = 0 carB = 1 : lightsA = 001 lightsB = 100 state =100000
reset = 0 carA = 0 carB = 1 : lightsA = 010 lightsB = 100 state =010000
reset = 0 carA = 0 carB = 1 : lightsA = 100 lightsB = 001 state =001000
reset = 0 carA = 0 carB = 1 : lightsA = 100 lightsB = 001 state =000100
reset = 0 carA = 0 carB = 0 : lightsA = 100 lightsB = 001 state =000100
reset = 0 carA = 1 carB = 0 : lightsA = 100 lightsB = 001 state =000100
reset = 0 carA = 1 carB = 0 : lightsA = 100 lightsB = 010 state =000010
```

```
reset = 0 carA = 1 carB = 0 : lightsA = 001 lightsB = 100 state =000001
reset = 0 carA = 1 carB = 0 : lightsA = 001 lightsB = 100 state =100000
$finish at simulation time                 2800
```

Prep Question 7 (Optional): Describe your traffic light controller in Verilog and simulate it.

Note that Verilog also allows you to describe the traffic light controller at a higher level, often called *behavioral* (as opposed to *structural*). For example, we can define the TLC module in behavioral Verilog without giving any logic equations as shown below. Its possible to write the model of a simple RISC microprocessor in about an evening using behavioral Verilog.

```
// tlc2.v
// traffic light controller in verilog - behavioral model
// W. J. Dally 10/31/2001
//

module TLC(clk, reset, carA, carB, lightsA, lightsB) ;

`define AG2 3'b000
`define AY 3'b001
`define BG1 3'b010
`define BG2 3'b110
`define BY 3'b101
`define AG1 3'b100

`define RED 3'b100
`define YELLOW 3'b010
`define GREEN 3'b001

  input clk ;                    // clock
  input reset ;                  // reset
  input carA ;                   // a car is waiting on A Street
  input carB ;                   // a car is waiting on B Street
  output[2:0] lightsA ;            // red, yellow, green for A
  output[2:0] lightsB ;            // red, yellow, green for B

  reg[2:0] state ;               // state vector

  // state function
  always @(posedge clk)
    begin
      if(reset) state = `AG2  ;
      else
        case(state)
          `AG2: state = carB ? `AY : `AG2 ;
          `AY: state = `BG1 ;
          `BG1: state = `BG2 ;
          `BG2: state = carA ? `BY : `BG2 ;
          `BY: state = `AG1 ;
          `AG1: state = `AG2 ;
        endcase
    end

  //output function
  assign lightsA = ((state == `AG2)|(state == `AG1)) ? `GREEN
              : (state == `AY) ? `YELLOW : `RED ;
  assign lightsB = ((state == `BG2)|(state == `BG1)) ? `GREEN
              : (state == `BY) ? `YELLOW : `RED ;

endmodule
```

We can simulate this behavioral model giving a result identical to the old model (except for the state vector).

```
reset = 1 carA = 0 carB = 0 : lightsA = xxx lightsB = xxx state = xxx
reset = 1 carA = 0 carB = 0 : lightsA = 001 lightsB = 100 state = 000
reset = 1 carA = 0 carB = 0 : lightsA = 001 lightsB = 100 state = 000
reset = 1 carA = 0 carB = 0 : lightsA = 001 lightsB = 100 state = 000
reset = 0 carA = 0 carB = 0 : lightsA = 001 lightsB = 100 state = 000
reset = 0 carA = 0 carB = 1 : lightsA = 001 lightsB = 100 state = 000
reset = 0 carA = 0 carB = 1 : lightsA = 010 lightsB = 100 state = 001
reset = 0 carA = 0 carB = 1 : lightsA = 100 lightsB = 001 state = 010
reset = 0 carA = 0 carB = 1 : lightsA = 100 lightsB = 001 state = 110
reset = 0 carA = 0 carB = 0 : lightsA = 100 lightsB = 001 state = 110
reset = 0 carA = 1 carB = 0 : lightsA = 100 lightsB = 001 state = 110
reset = 0 carA = 1 carB = 0 : lightsA = 100 lightsB = 010 state = 101
reset = 0 carA = 1 carB = 0 : lightsA = 001 lightsB = 100 state = 100
reset = 0 carA = 1 carB = 0 : lightsA = 001 lightsB = 100 state = 000
$finish at simulation time                 2800
```

We can also *synthesize* our behavioral Verilog, automatically generating the logic equations and mapping them onto a library of gates. Here is the result of synthesizing this model to cells from an ASIC gate library using the Synopsys design compiler. The TDN2Ks and the DTN2A are flip-flops, NA is NAND, NO NOR, IV Inverter and so on.

```
module TLC ( clk, reset, carA, carB, lightsA, lightsB );
output [2:0] lightsA;
output [2:0] lightsB;
input  clk, reset, carA, carB;
    wire \state[2] , \n144[0] , \state143[1] , \state[1] , \state[0] ,
        \state143[0] , n551, n552, n553, n554, n555, n556, n557, n558, n559,
        n560, n561, n562, n563, n564, n565, n566, n567;
    IV110 U169 ( .A(n551), .Y(n567) );
    IV110 U170 ( .A(carA), .Y(n564) );
    MU112 U171 ( .A(n565), .B(n566), .S(\state[2] ), .Y(\state143[0] ) );
    IV110 U172 ( .A(n559), .Y(n561) );
    IV110 U173 ( .A(n562), .Y(lightsB[1]) );
    IV110 U174 ( .A(n563), .Y(lightsB[0]) );
    IV110 U175 ( .A(\state[0] ), .Y(n557) );
    IV110 U176 ( .A(n555), .Y(lightsA[1]) );
    IV110 U177 ( .A(n556), .Y(lightsA[0]) );
    IV110 U178 ( .A(\state[2] ), .Y(n558) );
    DTN2A \state_reg[1]  ( .CLK(clk), .D(\state143[1] ), .ENZ(n552), .Q(
        \state[1] ) );
    BU1D0 U179 ( .A(reset), .Y(n551) );
    TO010 U180 ( .HI(n553), .LO(n554) );
    TDN2K \state_reg[2]  ( .CLK(clk), .D(\state[1] ), .ENZ(\state[0] ), .SCAN(
        n551), .SD(n554), .Q(\state[2] ) );
    IV110 U181 ( .A(\n144[0] ), .Y(n552) );
    TDN2K \state_reg[0]  ( .CLK(clk), .D(n554), .ENZ(n561), .SCAN(
        \state143[0] ), .SD(n553), .Q(\state[0] ) );
    AN210 U182 ( .A(n555), .B(n556), .Y(lightsA[2]) );
    BF053 U183 ( .A1(carA), .A2(n559), .B1(lightsA[0]), .B2(n560), .Y(
        \state143[1] ) );
    NA211 U184 ( .A(n557), .B(n561), .Y(\n144[0] ) );
    AN210 U185 ( .A(n562), .B(n563), .Y(lightsB[2]) );
    NA211 U186 ( .A(n551), .B(\state[1] ), .Y(n559) );
    NA211 U187 ( .A(\state[1] ), .B(n557), .Y(n556) );
    NA211 U188 ( .A(\state[2] ), .B(n567), .Y(n560) );
    NA211 U189 ( .A(\state[0] ), .B(\state[1] ), .Y(n563) );
    NA311 U190 ( .A(\state[1] ), .B(n558), .C(\state[0] ), .Y(n555) );
    NA311 U191 ( .A(\state[1] ), .B(\state[2] ), .C(\state[0] ), .Y(n562) );
    NA211 U192 ( .A(n564), .B(n561), .Y(n566) );
    NA311 U193 ( .A(reset), .B(lightsA[0]), .C(carB), .Y(n565) );
endmodule
```

Prep Question 8 (Optional): Describe your traffic light controller in behavioral Verilog and synthesize it.