

---

# Digital Design: A Systems Approach

## Lecture 6: Sequential Logic

# Readings

---

- L6: Chapter 14
- L7: Chapter 16

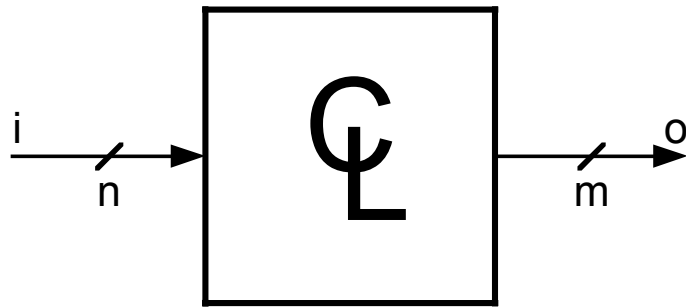
# Today

---

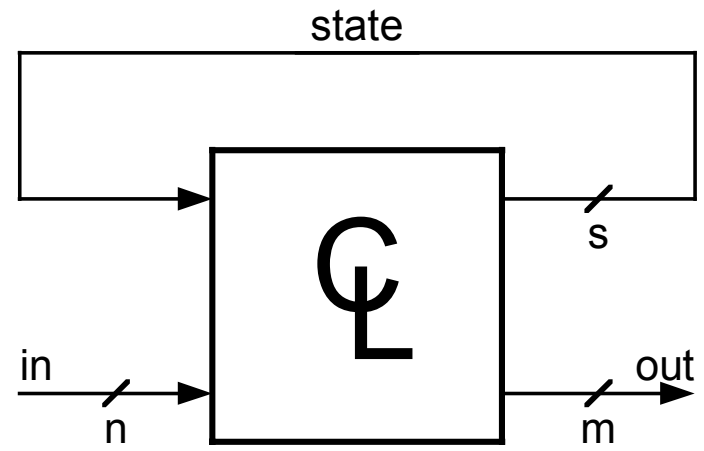
- Sequential logic: circuits with state
  - *Synchronizes* state transitions to a clock
- D Flip-flops (DFF)
- Finite state machines (FSMs)
  - One-hot
  - Binary
- Designing state machines
  - State/input tables
- FSMs in Verilog
  - Style
  - Explicit state declaration

# Sequential logic has state

---



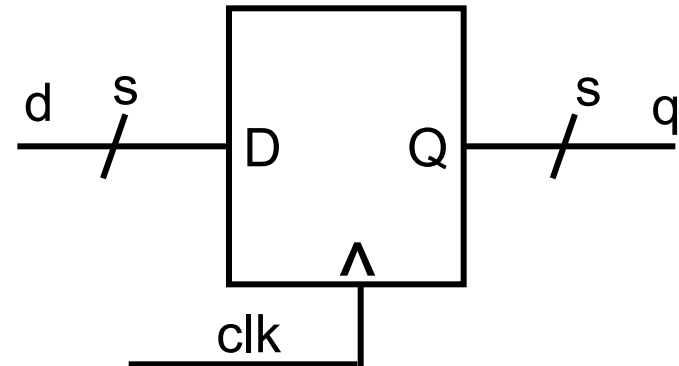
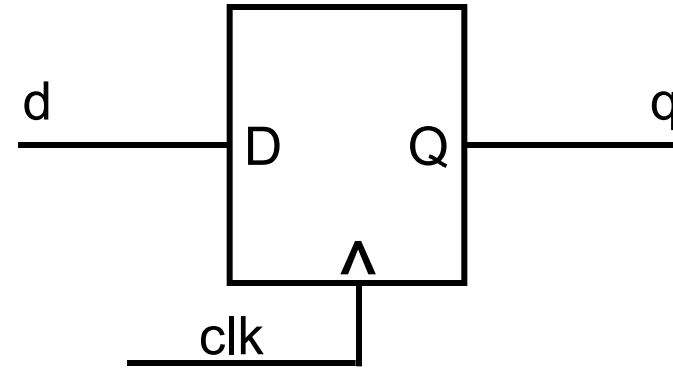
Combinational Logic



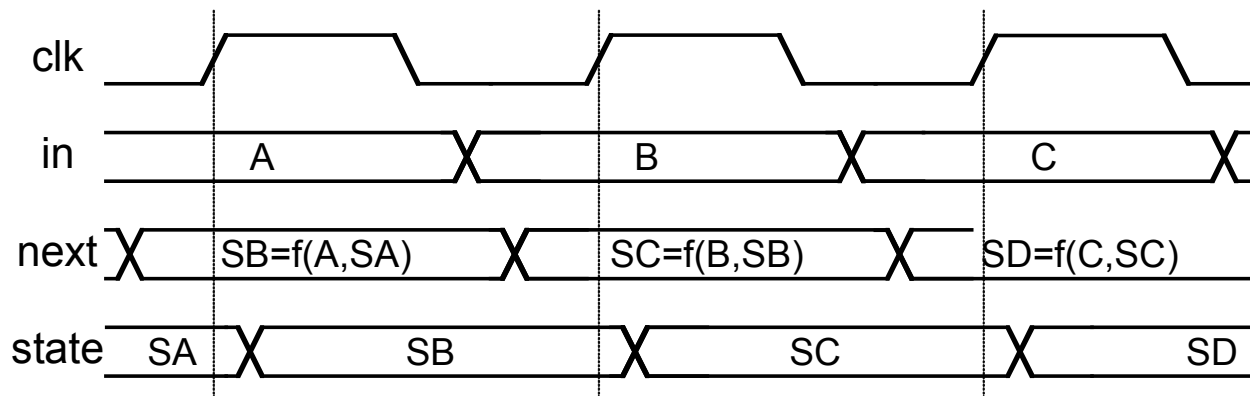
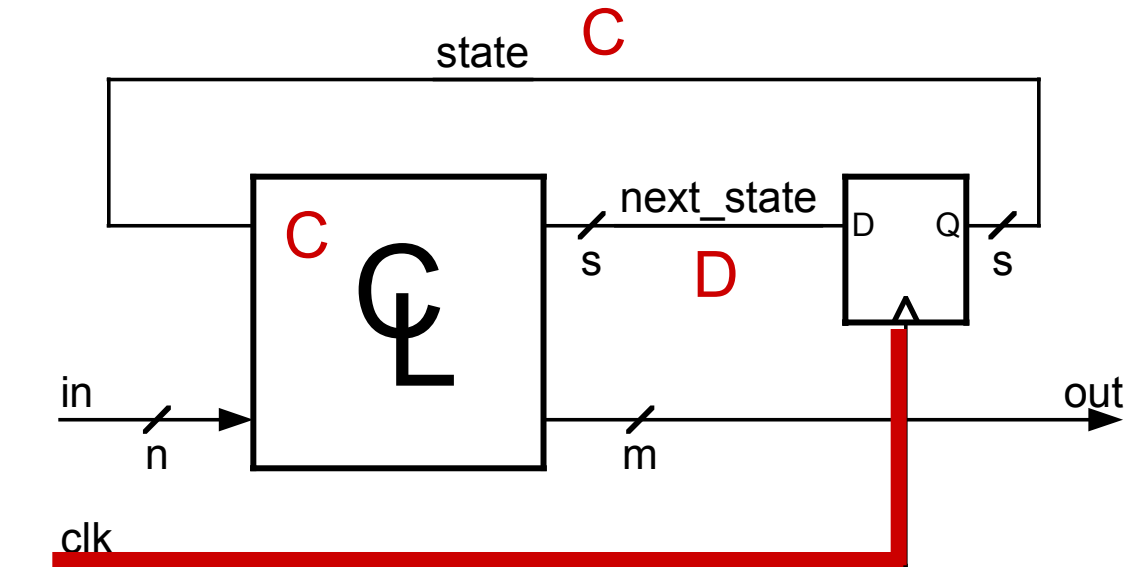
Sequential Logic

# D Flip-Flop

- Input: D
- Output: Q
- Clock
- Q outputs a steady value
- Edge on Clock changes Q to be D
- Flip-flop stores state
- Allows sequential circuits to iterate

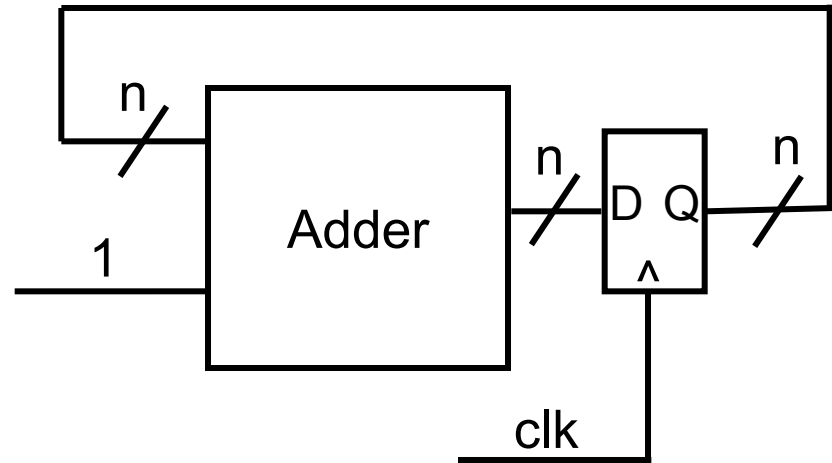


# Synchronous Sequential Logic



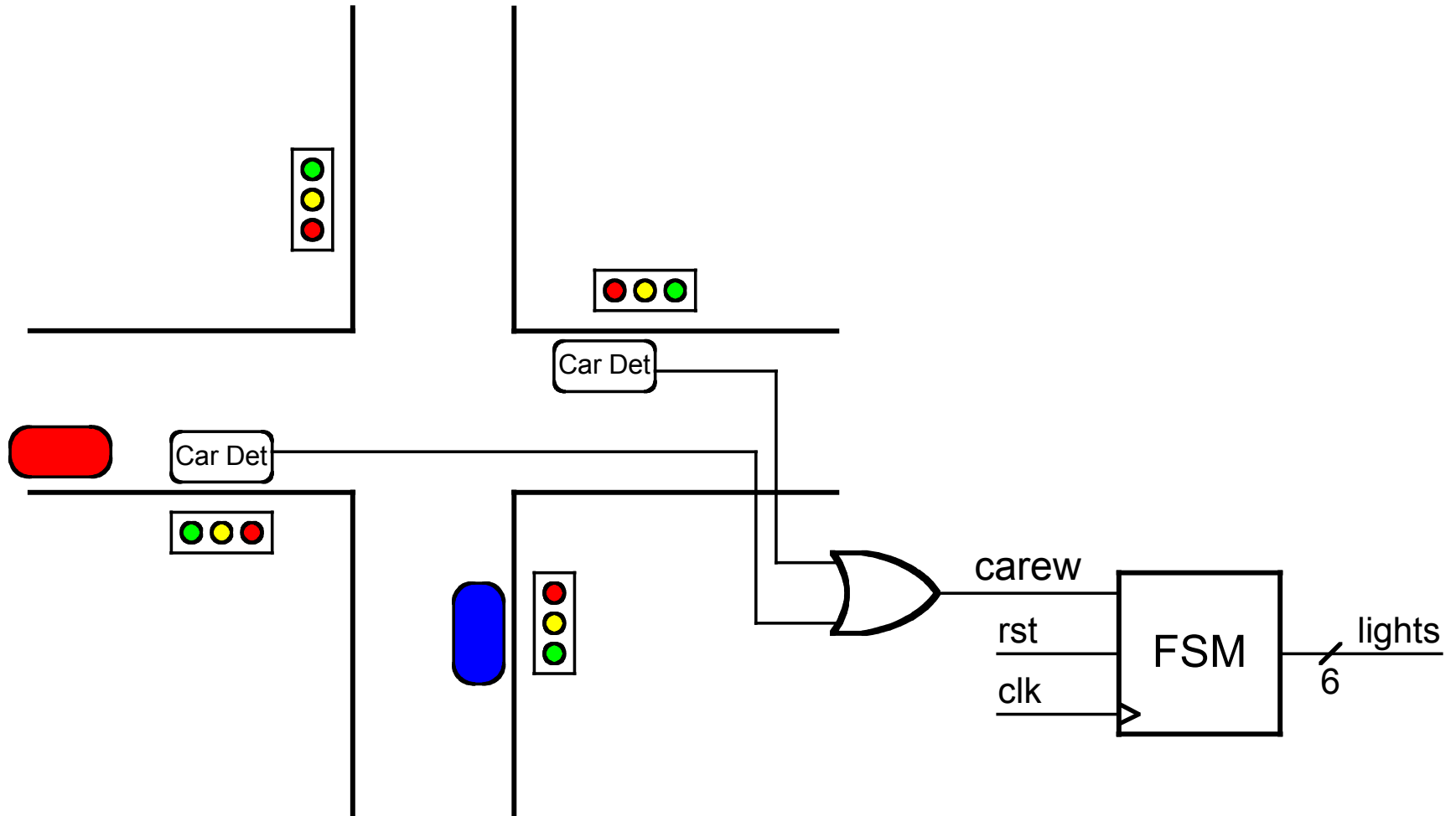
## Example Sequential Circuit: Increment

- Increment a count on every clock tick
- Just wrap around on overflow



- Maintaining state has a lot more uses than arithmetic

# Example: A Traffic-Light Controller





# Traffic Light FSM Behavior

---

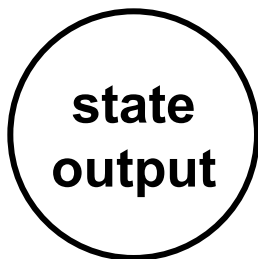
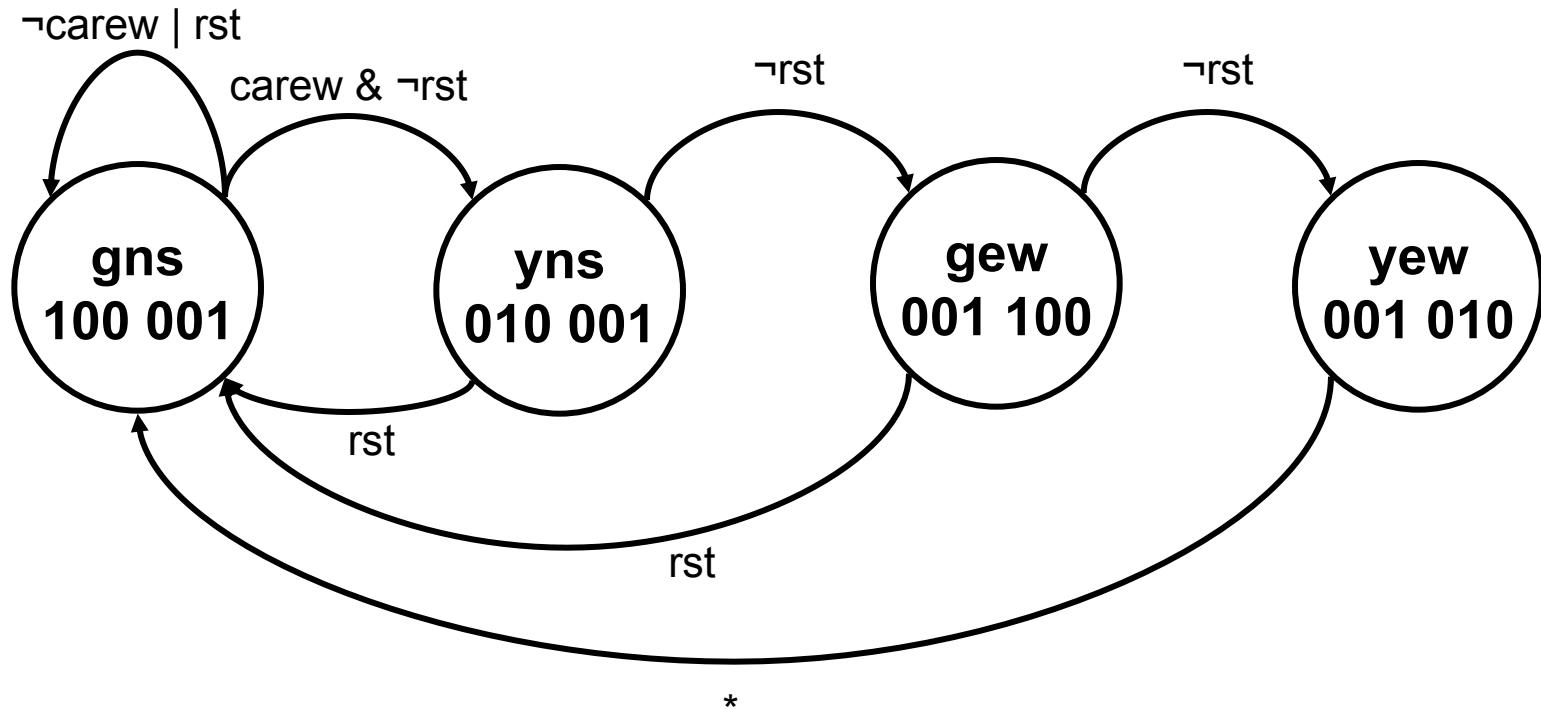
- Reset to green in north-south direction
- If light is green or yellow in one direction, it must be red in the other
- A light must be yellow between changing from green to red
- If there is a car waiting east-west, then east-west should become green before returning to green north-south

# Traffic Light FSM

---

- Finite state machine is specified by
  - States, Inputs, Outputs
- Four states
  - Green north-south (red east-west)
  - Yellow north-south (red east-west)
  - Green east-west (red north-south)
  - Yellow east-west (red north-south)
- Input
  - Carew: is there a car waiting on the east-west road
  - Reset: return to initial state (need not go through yellow)
- Output: Two 3-bit traffic light signals (1-hot) (gyr gyr)
  - 100 001: NS green, EW red; 001 010 NS red, EW yellow

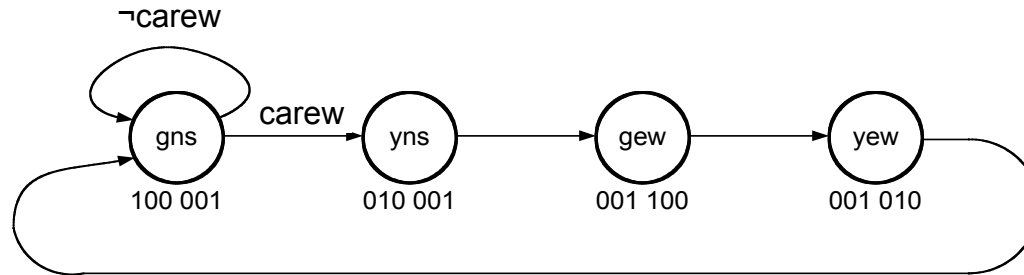
# Complete Traffic Light State machine



= gyr gyr  
ns ew

carew = car east-west sensor active  
rst = reset

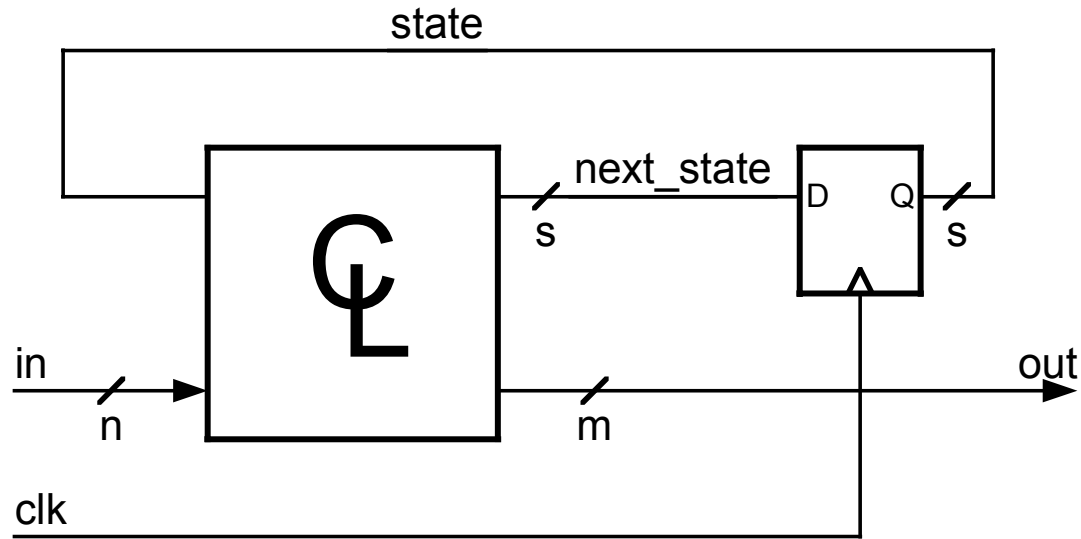
## State Table (ignoring reset for now)



| Current State | Next State          |                | Output |      |
|---------------|---------------------|----------------|--------|------|
|               | $\neg \text{carew}$ | $\text{carew}$ |        |      |
| gns           | gns                 | yns            | 100001 | lgns |
| yns           | gew                 | gew            | 010001 | lyns |
| gew           | yew                 | yew            | 001100 | lgew |
| yew           | gns                 | gns            | 001010 | lyew |

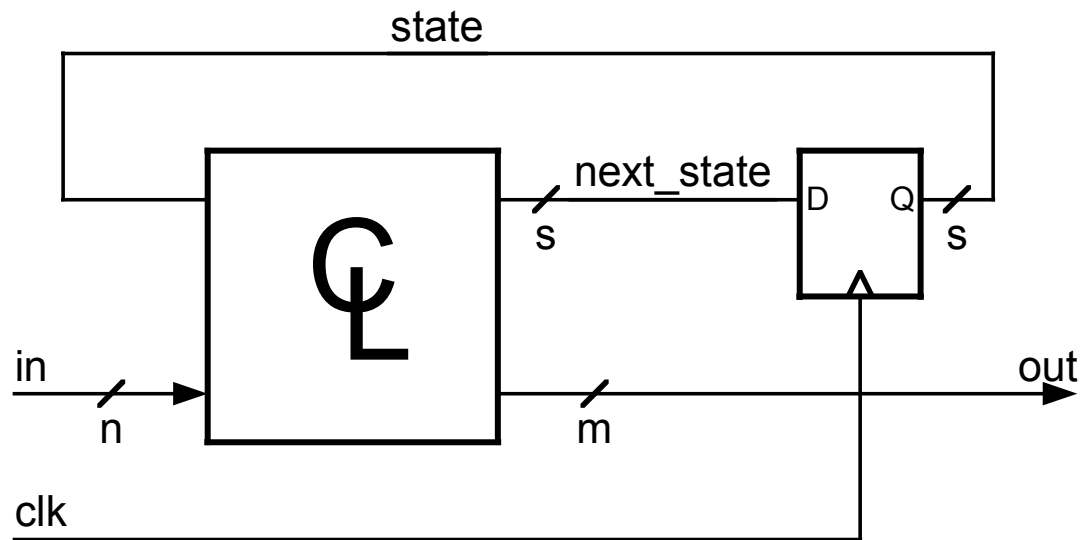
# State Assignment

- Assign values to encode each state (gns, yns, gew, yew) on state and next\_state signals

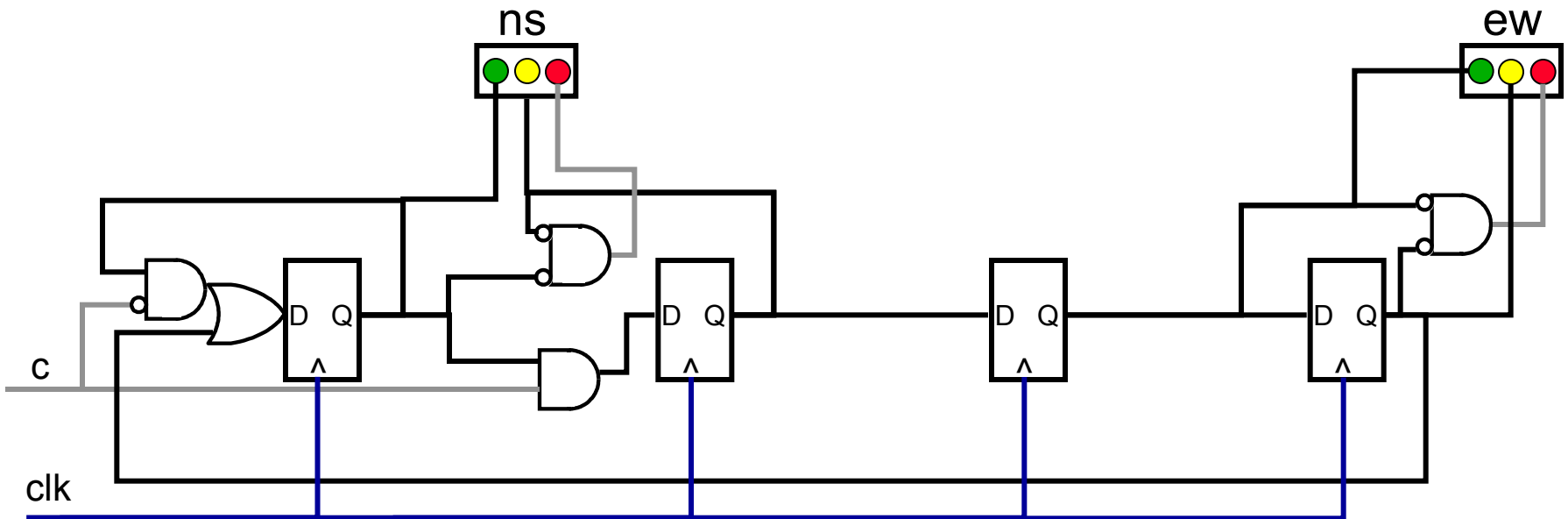
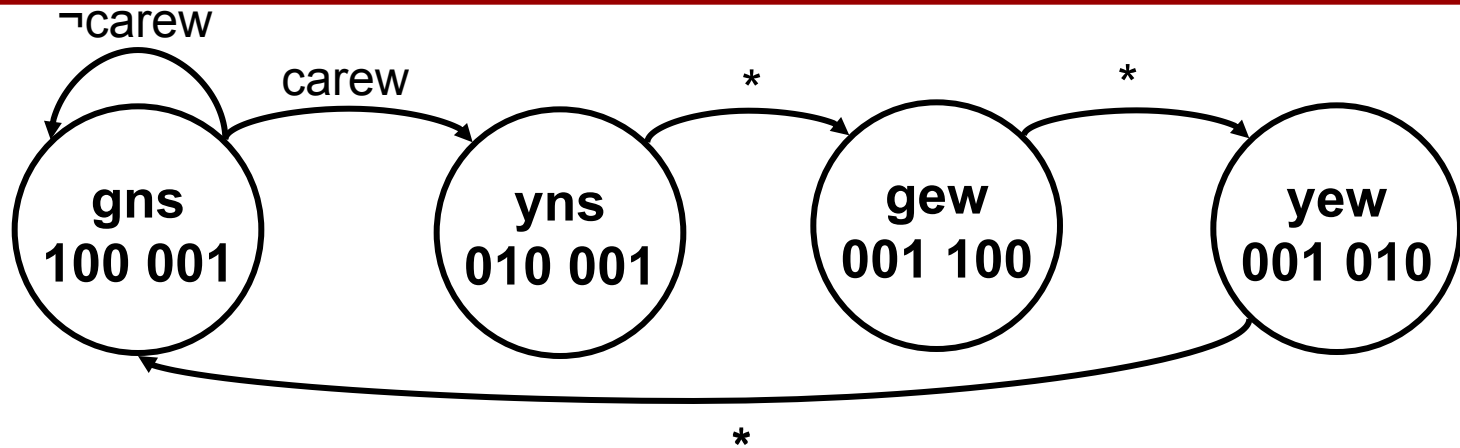


# One-Hot State Assignment

| State | Encoding |
|-------|----------|
| gns   | 1000     |
| yns   | 0100     |
| gew   | 0010     |
| yew   | 0001     |

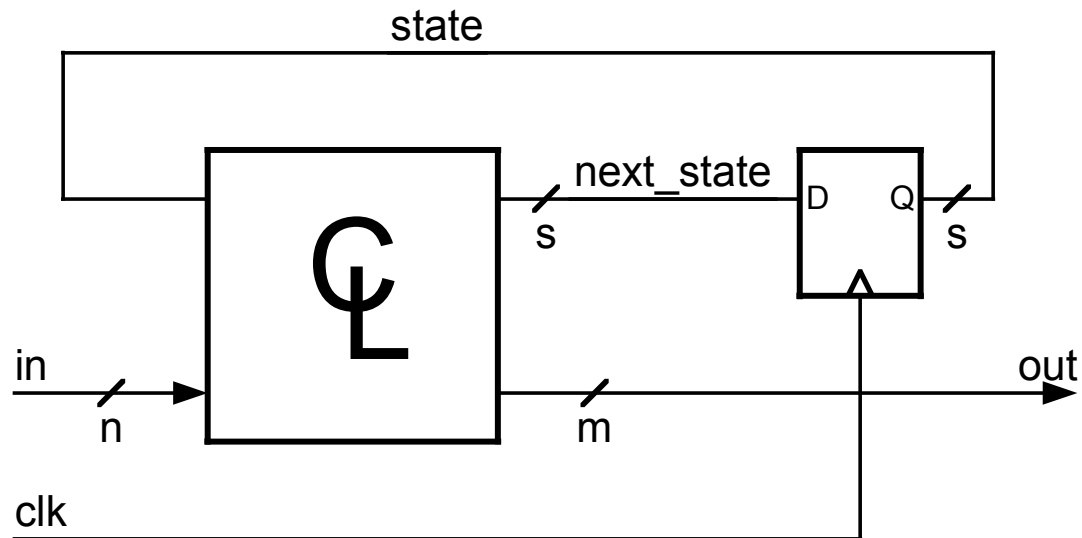


# Translating No-Reset FSM To One-Hot Controller



# Binary State Assignment (Gray Code)

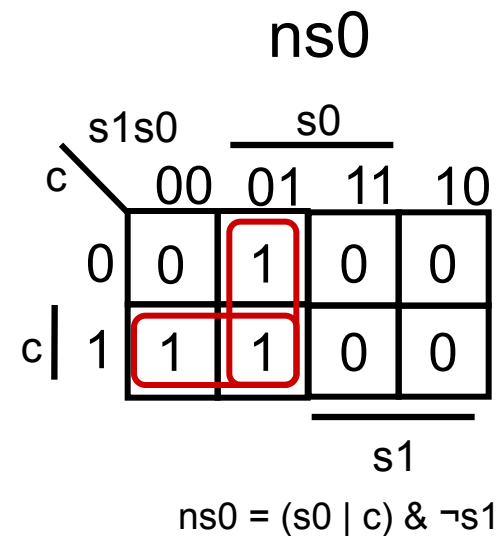
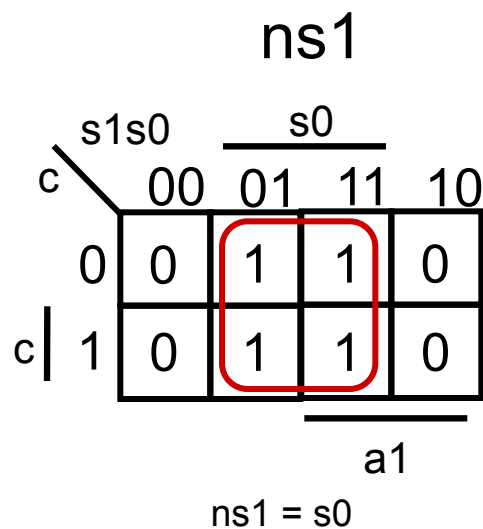
| State | Encoding |
|-------|----------|
| gns   | 00       |
| yns   | 01       |
| gew   | 11       |
| yew   | 10       |





# Encoded state table and next-state K-maps

| State | Next State          |                | Output |
|-------|---------------------|----------------|--------|
|       | $\neg \text{carew}$ | $\text{carew}$ |        |
| 00    | 00                  | 01             | 100001 |
| 01    | 11                  | 11             | 010001 |
| 11    | 10                  | 10             | 001100 |
| 10    | 00                  | 00             | 001010 |



# Logic Equations

---

| State | Next State         |                | Output |
|-------|--------------------|----------------|--------|
|       | $\neg\text{carew}$ | $\text{carew}$ |        |
| 00    | 00                 | 01             | 100001 |
| 01    | 11                 | 11             | 010001 |
| 11    | 10                 | 10             | 001100 |
| 10    | 00                 | 00             | 001010 |

$$\text{ns1} = \text{s0}$$

$$\text{ns0} = (\text{s0} \mid \text{carew}) \& \neg\text{s1}$$

$$\text{lgns} = \neg\text{s0} \& \neg\text{s1}$$

$$\text{lyns} = \text{s0} \& \neg\text{s1}$$

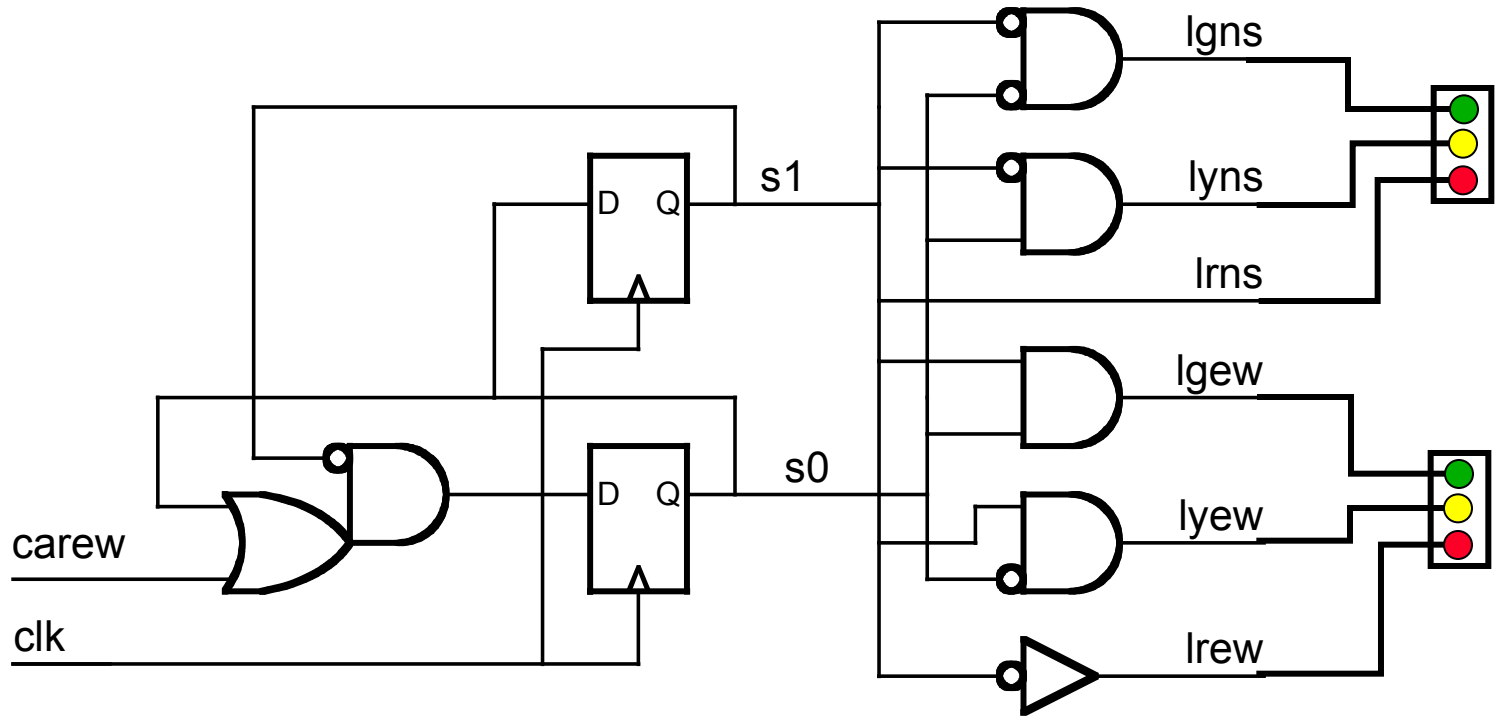
$$\text{lrns} = \text{s1}$$

$$\text{lgew} = \text{s0} \& \text{s1}$$

$$\text{lyew} = \neg\text{s0} \& \text{s1}$$

$$\text{lrw} = \neg\text{s1}$$

# Implementation of traffic-light controller with binary state assignment



## Another Example - Divide by 3 FSM

---

- Output: (out) goes high once for each third cycle that input (in) is high
- Example
  - IN:        0101010011100
  - OUT:      0000001000010
  - Note one cycle delay

## Another Example - Divide by 3 FSM

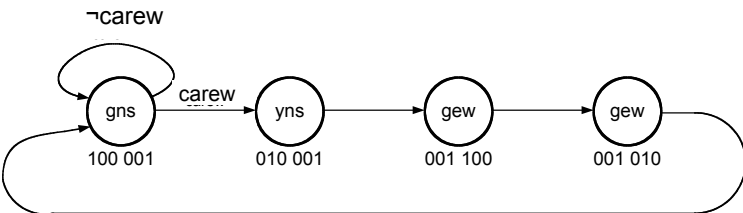
---

- Output: (out) goes high once for each third cycle that input (in) is high
- Example
  - IN:        0101010011100
  - OUT:      0000001000010
  - Note one cycle delay
- Draw state diagram
  - How many states do we need?
  - What are the transitions?
  - What are the outputs?
- How do we make the output go high in the same cycle as the third input 1?

## FSMs in Verilog (Style guide)

---

- All feedback through explicitly instantiated DFF module!
- Next state and output functions are combinational – case or assign
  - No “inferred latches”
- Make sure you can reset your FSM
- Use defines for state encoding (and width)



```

//-----
// FSM Example for Lecture 7
// Bill Dally 1/30/03
//-----
// define state assignment - one hot
//-----
`define SWIDTH 4
`define GNS 4'b1000
`define YNS 4'b0100
`define GEW 4'b0010
`define YEW 4'b0001
//-----
// define output codes
//-----
`define GNSL 6'b100001
`define YNSL 6'b010001
`define GEWL 6'b001100
`define YEWL 6'b001010
//-----
// define flip-flop
//-----
module DFF(clk, in, out) ;
  parameter n = 1; // width
  input clk ;
  input [n-1:0] in ;
  output [n-1:0] out ;
  reg [n-1:0] out ;

  always @(posedge clk)
    out = in ;
endmodule

```

```

//-----
// Traffic_Light
// Inputs:
//   clk - system clock
//   rst - reset - high true
//   carew - car east/west - true when car is waiting in east-west direction
// Outputs:
//   lights - (6 bits) {gns, yns, rns, gew, yew, rew}
// Waits in state GNS until carew is true, then sequences YNS, GEW, YEW
// and back to GNS.
//-----
module Traffic_Light(clk, rst, carew, lights) ;
  input clk ;
  input rst ; // reset
  input carew ; // car present on east-west road
  output [5:0] lights ; // {gns, yns, rns, gew, yew, rew}
  wire [SWIDTH-1:0] state, next ; // current and next state
  reg [SWIDTH-1:0] next1 ; // next state w/o reset
  reg [5:0] lights ; // output - six lights 1=on

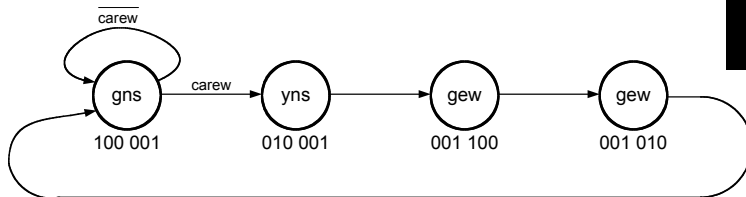
  // instantiate state register
  DFF #(SWIDTH) state_reg(clk, next, state) ;

  // next state and output equations - this is combinational logic
  always @(state or carew) begin
    case(state)
      `GNS: {next1, lights} = {(carew ? `YNS : `GNS), `GNSL} ;
      `YNS: {next1, lights} = `{GEW, `YNSL} ;
      `GEW: {next1, lights} = `{YEW, `GEWL} ;
      `YEW: {next1, lights} = `{GNS, `YEWL} ;
    endcase
  end
  // add reset
  assign next = rst ? `GNS : next1 ;
endmodule

```

# Feedback (state) in explicitly declared DFF module

```
//-----  
// FSM Example for Lecture 7  
// Bill Dally 1/30/03  
//-----  
// define state assignment - one hot  
//-----  
`define SWIDTH 4  
`define GNS 4'b1000  
`define YNS 4'b0100  
`define GEW 4'b0010  
`define YEW 4'b0001  
//-----  
// define output codes  
//-----  
`define GNSL 6'b100001  
`define YNSL 6'b010001  
`define GEWL 6'b001100  
`define YEWL 6'b001010  
//-----  
// define flip-flop  
//-----  
module DFF(clk, in, out) ;  
    parameter n = 1; // width  
    input clk ;  
    input [n-1:0] in ;  
    output [n-1:0] out ;  
    reg [n-1:0] out ;  
  
    always @(posedge clk)  
        out = in ;  
endmodule
```

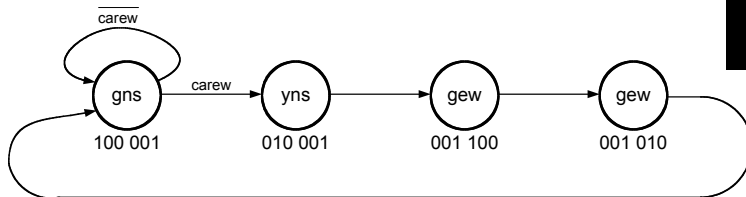




# Next-state and output functions are combinational

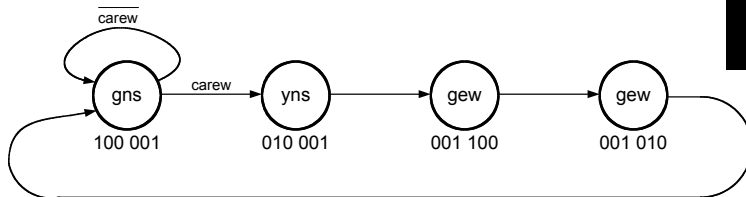
```
//-----  
// FSM Example for Lecture 7  
// Bill Dally 1/30/03  
//-----  
// define state assignment - one hot  
//-----  
`define SWIDTH 4  
`define GNS 4'b1000  
`define YNS 4'b0100  
`define GEW 4'b0010  
`define YEW 4'b0001  
//-----  
// d  
//--  
`def  
`def  
`def  
`def  
//--  
// d  
//--  
module  
  parameter n = 1, // width  
  input clk ;  
  input [n-1:0] in ;  
  output [n-1:0] out ;  
  reg [n-1:0] out ;  
  
  always @(posedge clk)  
    out = in ;  
endmodule
```

**Case statement  
concatenation  
to assign next state  
and outputs**



# Explicit reset

```
//-----  
// FSM Example for Lecture 7  
// Bill Dally 1/30/03  
//-----  
// define state assignment - one hot  
//-----  
'define SWIDTH 4  
'define GNS 4'b1000  
'define YNS 4'b0100  
'define GEW 4'b0010  
'define YEW 4'b0001  
//-----  
// define output codes  
//-----  
'define GNSL 6'b100001  
'define YNSL 6'b010001  
'define GEWL 6'b001100  
'define YEWL 6'b001010  
//-----  
// define flip-flop  
//-----  
module DFF(clk, in, out) ;  
    parameter n = 1; // width  
    input clk ;  
    input [n-1:0] in ;  
    output [n-1:0] out ;  
    reg [n-1:0] out ;  
  
    always @(posedge clk)  
        out = in ;  
endmodule
```

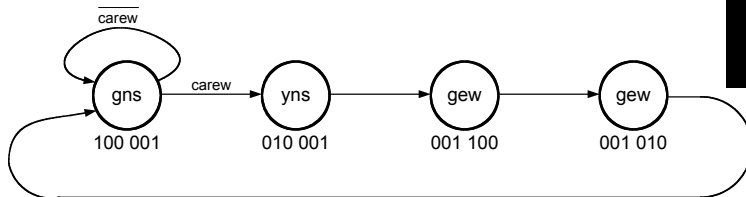


**Separate assign  
for reset**

# Use defines for state assignment and width

## Defines for output codes are a good idea too.

```
//-----  
// FSM Example for Lecture 7  
// Bill Dally 1/30/03  
//-----  
// define state assignment - one hot  
//-----  
`define SWIDTH 4  
`define GNS 4'b1000  
`define YNS 4'b0100  
`define GEW 4'b0010  
`define YEW 4'b0001  
//-----  
// define output codes  
//-----  
`define GNSL 6'b100001  
`define YNSL 6'b010001  
`define GEWL 6'b001100  
`define YEWL 6'b001010  
//-----  
// define flip-flop  
//-----  
module DFF(clk, in, out) ;  
    parameter n = 1; // width  
    input clk ;  
    input [n-1:0] in ;  
    output [n-1:0] out ;  
    reg [n-1:0] out ;  
  
    always @(posedge clk)  
        out = in ;  
endmodule
```



```

# 0 x xxxx xxxxxx
# 1 0 xxxx xxxxxx
# 0 0 1000 100001
# 0 0 1000 100001
# 0 0 1000 100001
# 0 1 1000 100001
# 0 0 0100 010001
# 0 0 0010 001100
# 0 0 0001 001010
# 0 0 1000 100001
# 0 0 1000 100001
# 0 0 1000 100001
# 0 1 1000 100001
# 0 1 0100 010001
# 0 1 0010 001100
# 0 1 0001 001010
# 0 1 1000 100001
# 0 1 0100 010001

```

```

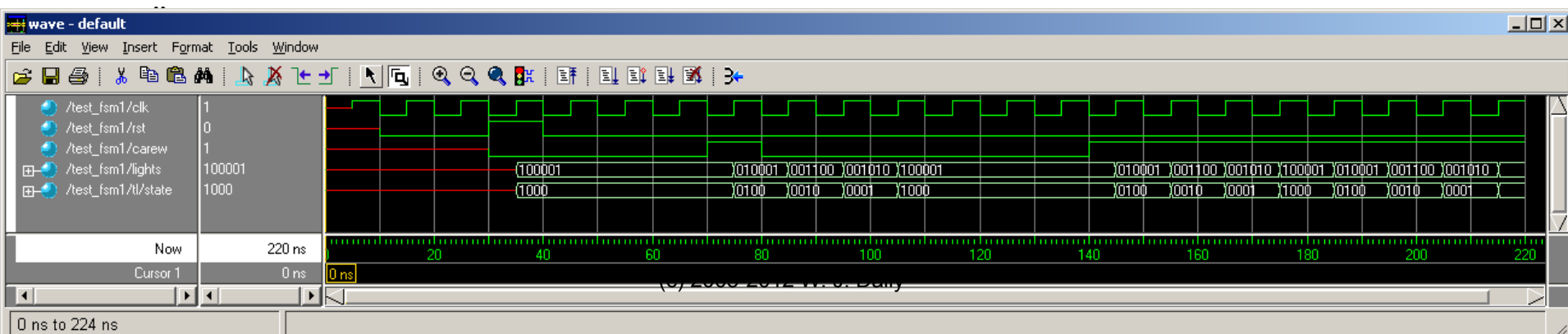
//-----
// test bench
//-----
module test_fsm1 ;
    reg clk, rst, carew ;
    wire [5:0] lights ;

    traffic_light tl(clk, rst, carew, lights) ;

    // clock with period of 10 units
    initial
        forever
            begin
                #5 clk = 1 ; #5 clk = 0 ;
                $display("%b %b %b %b",
                    rst, carew, tl.state, lights ) ;
            end

    // input stimuli
    initial begin
        #10 rst = 0 ; // start w/o reset to show x state
        #20 rst = 1 ; carew = 0 ; // reset
        #10 rst = 0 ; // remove reset
        #30 carew = 1 ; // wait 3 cycles, then car arrives
        #10 carew = 0 ; // car leaves
        #60 carew = 1 ; // wait 6 cycles then car comes and stays
        #80 // 8 more cycles
        $stop ;
    end
endmodule

```



```

# 0 x xxxx xxxxxx
# 1 0 xxxx xxxxxx
# 0 0 1000 100001
# 0 0 1000 100001
# 0 0 1000 100001
# 0 1 1000 100001
# 0 0 0100 010001
# 0 0 0010 001100
# 0 0 0001 001010
# 0 0 1000 100001
# 0 0 1000 100001
# 0 0 1000 100001
# 0 1 1000 100001
# 0 1 0100 010001
# 0 1 0010 001100
# 0 1 0001 001010
# 0 1 1000 100001
# 0 1 0100 010001

```

**State starts at X  
need to reset**

```

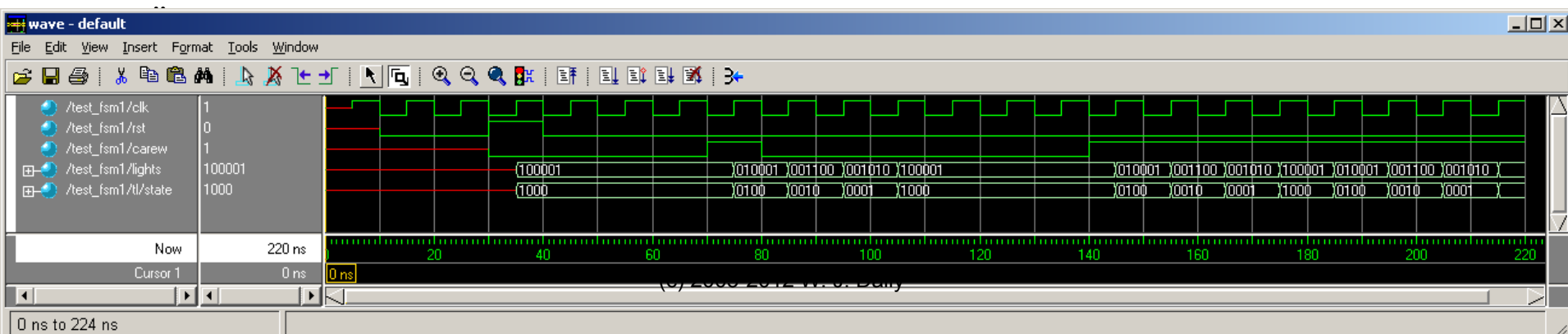
reg clk, rst, carew,
wire [5:0] lights ;

traffic_light tl(clk, rst, carew, lights) ;

// clock with period of 10 units
initial
  forever
    begin
      #5 clk = 1 ; #5 clk = 0 ;
      $display("%b %b %b %b",
        rst, carew, tl.state, lights ) ;
    end

// input stimuli
initial begin
  #10 rst = 0 ; // start w/o reset to show x state
  #20 rst = 1 ; carew = 0 ; // reset
  #10 rst = 0 ; // remove reset
  #30 carew = 1 ; // wait 3 cycles, then car arrives
  #10 carew = 0 ; // car leaves
  #60 carew = 1 ; // wait 6 cycles then car comes and stays
  #80 // 8 more cycles
  $stop ;
end
endmodule

```



```

# 0 x xxxx xxxxxx
# 1 0 xxxx xxxxxx
# 0 0 1000 100001
# 0 0 1000 100001
# 0 0 1000 100001
# 0 1 1000 100001
# 0 0 0100 010001
# 0 0 0010 001100
# 0 0 0001 001010
# 0 0 1000 100001
# 0 0 1000 100001
# 0 0 1000 100001
# 0 1 1000 100001
# 0 1 0100 010001
# 0 1 0010 001100
# 0 1 0001 001010
# 0 1 1000 100001
# 0 1 0100 010001

```

```

//-----
// test bench
//-----
module test_fsm1 ;
    reg clk, rst, carew ;
    wire [5:0] lights ;

```

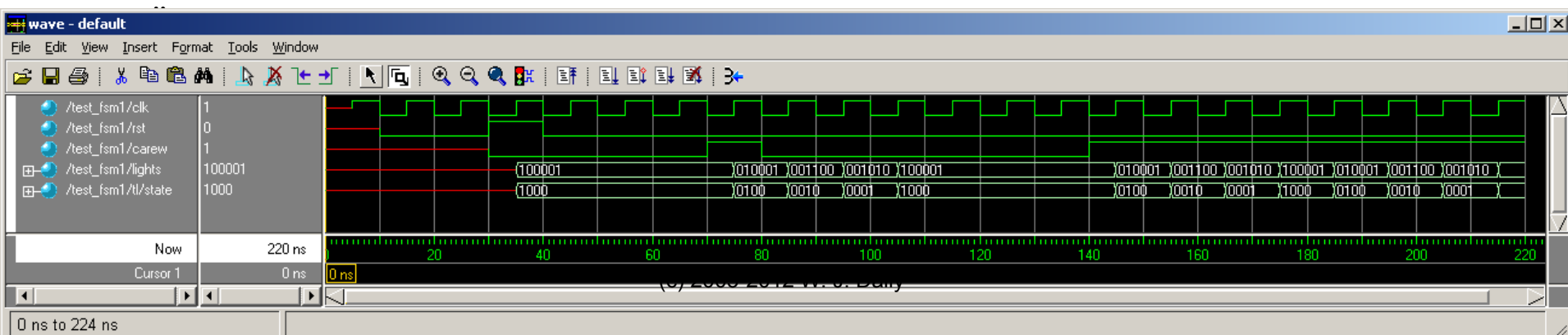
**Car arrives on  
east-west road**

```

begin
    #5 clk = 1 ; #5 clk = 0 ;
    $display("%b %b %b %b",
        rst, carew, tl.state, lights ) ;
end

// input stimuli
initial begin
    #10 rst = 0 ; // start w/o reset to show x state
    #20 rst = 1 ; carew = 0 ; // reset
    #10 rst = 0 ; // remove reset
    #30 carew = 1 ; // wait 3 cycles, then car arrives
    #10 carew = 0 ; // car leaves
    #60 carew = 1 ; // wait 6 cycles then car comes and stays
    #80 // 8 more cycles
    $stop ;
end
endmodule

```



# Change the state assignment with defines

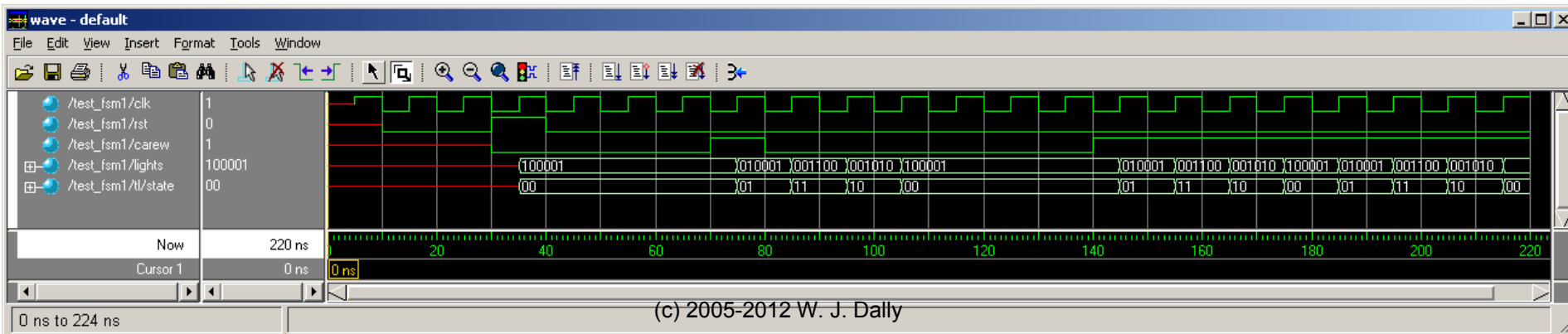
---

```
//-----  
// define state assignment - Gray code  
//-----  
`define SWIDTH 2  
`define GNS 2'b00  
`define YNS 2'b01  
`define GEW 2'b11  
`define YEW 2'b10
```

```

# 0 x xx xxxxxx
# 1 0 xx xxxxxx
# 0 0 00 100001
# 0 0 00 100001
# 0 0 00 100001
# 0 1 00 100001
# 0 0 01 010001
# 0 0 11 001100
# 0 0 10 001010
# 0 0 00 100001
# 0 0 00 100001
# 0 0 00 100001
# 0 1 00 100001
# 0 1 01 010001
# 0 1 11 001100
# 0 1 10 001010
# 0 1 00 100001
# 0 1 01 010001

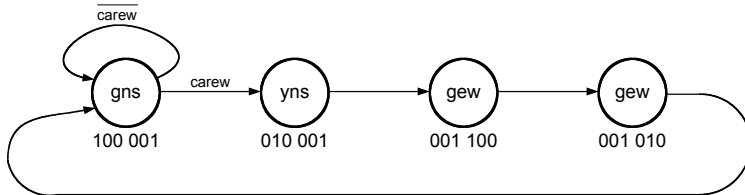
```





# Summary

- Sequential logic
  - State feedback around combinational logic
- Synchronous sequential logic
  - State register *synchronizes* state transitions with a clock
  - All bits of state are updated at the same time on the rising edge of the clock
  - Reduces synchronous machine design to combinational design of next-state function
- State diagram and table describe next-state and output functions



| Current State | Next State |       | Output |      |
|---------------|------------|-------|--------|------|
|               | lcarew     | carew |        |      |
| gns           | gns        | yns   | 100001 | lgns |
| yns           | gew        | gew   | 010001 | lyns |
| gew           | yew        | yew   | 001100 | lgew |
| yew           | gns        | gns   | 001010 | lyew |

- State assignment
  - Assign binary codes to states – one hot or binary
- Implementation
  - Design combinational logic for next-state and output functions
- Verilog
  - Explicitly instantiate flip-flops
  - Case (or casex) statement for next-state function
  - Use defines for state assignment

## Next Time

---

- Data Path FSMs