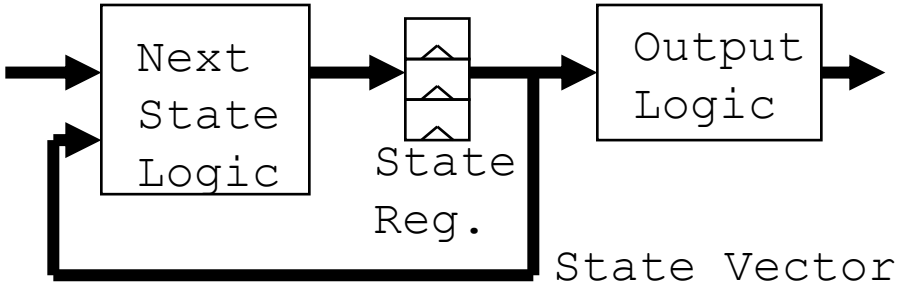
Finite State Machines

Reset

- Reset is a global signal that the designer can not modify
- It is generally asserted on power up or a “hard” reset
- It is used to start the machine in a “known” state
- Thus it must be distributed to
  - All FSMs
  - Selected counters
  - Selected status registers

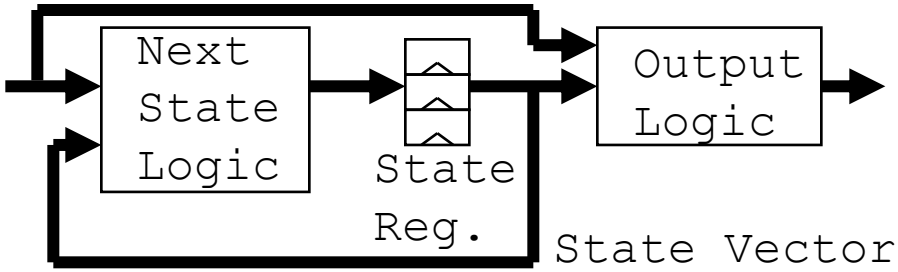
Finite State Machine Types

- Finite State Machines can be classified by the following attributes:
- Moore or Mealy type outputs



Moore Outputs

Outputs depend solely on state vector (generally, a Moore FSM is the simplest to design)



Mealy Outputs

Outputs depend on inputs and state vector (only use if it is significantly smaller or faster)

… FSM Types

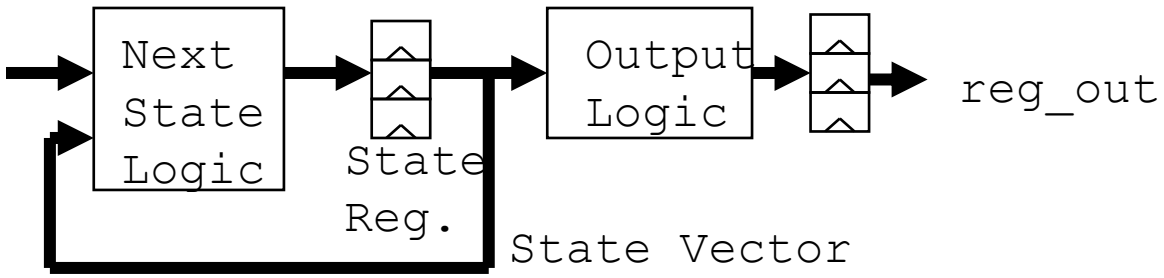
- State Vector Encoding
  - Minimal encoding
    - Minimum number of bits
  - Minimum, sequential encoding
    - Minimum number of bits and states in sequence
      - Does not necessarily optimize ‘next state logic’ size
  - Gray encoding
    - state bit changes by only one bit between sequential states
      - Minimizes switching activity in state vector register
  - One-hot encoding
    - one bit per state
      - usually gives fastest ‘next state’ logic
- Example: 7-state FSM, states S0 … S7:

… FSM Types

- Resets:
  - Reset usually occurs only on power-up and when someone hits the ‘reset’ button
  - Asynchronous reset:
    - FSM goes to reset state whenever reset occurs
  - Synchronous reset:
    - FSM goes to reset state on the next clock edge after reset occurs
  - Asynchronous reset leads to smaller flip-flops while synchronous reset is ‘safer’ (noise on the reset line is less likely to accidently cause a reset).
- Fail-Safe Behavior:
  - If the FSM enters an ‘illegal’ state due to noise is it guaranteed to then enter a legal state?
    - ‘Yes’ is generally desirable

… FSM Types

- Sequential Next state or output logic
  - Usually, these blocks are combinational logic only
  - However, can place sequential logic (e.g. a counter, or a toggle-flip-flop) in these blocks if it is advantageous
  - AVOID DOING THIS AS MUCH AS YOU CAN UNLESS YOU ARE REALLY SURE ABOUT WHAT YOU ARE DOING
    - Sequential next state or output logic can get very confusing to design and debug
- Registered or Unregistered Outputs
  - Do not register the outputs unless you need to 'deglitch' the outputs (for example, for asynchronous handshaking - combinational logic has to be assumed to be glitchy) or are pipelining the control
  - e.g.

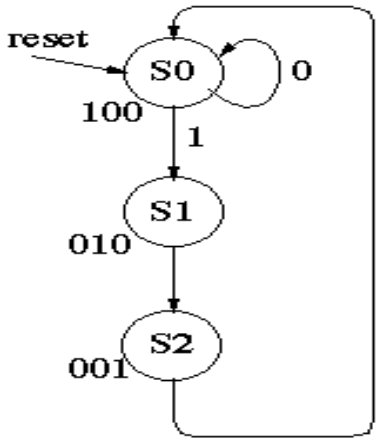


Example - Drag Racing Lights

- At the start of a new race (‘car’), go through the Red-Yellow-Green sequence:

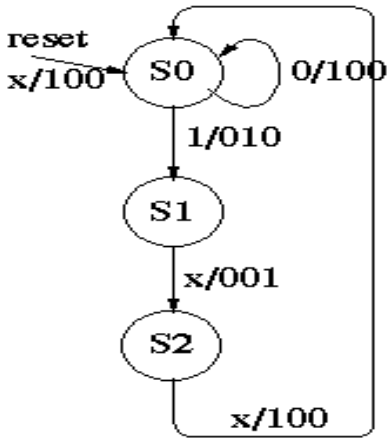
Moore Machine:

Nomenclature: inputs
car?
On states: red yellow green



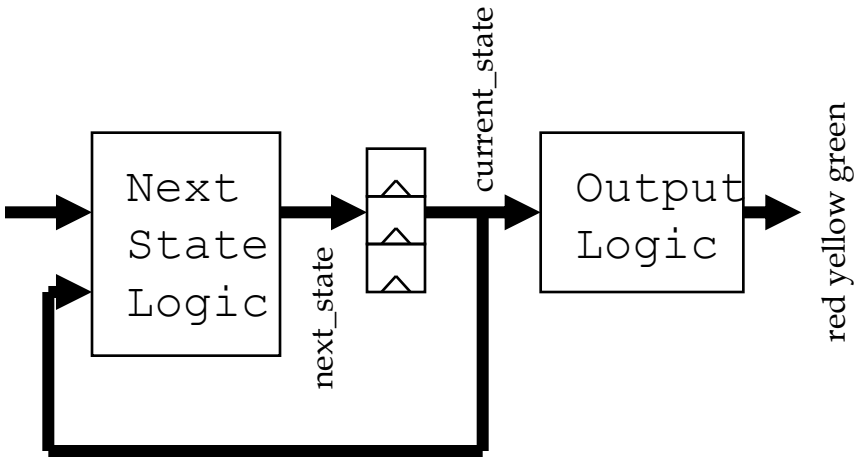
Mealy Machine:

Nomenclature: inputs / outputs
car? / red yellow green



Drag Light Controller ...Verilog

```
module traffic_light_controller (clock, reset, car, red, yellow, green);

input clock;
input reset;
input car;
output red, yellow, green;

parameter [1:0] // synopsys enum states
S0 = 2'b00,
S1 = 2'b01,
S2 = 2'b10,
S3 = 2'b11;
reg [1:0] /* synopsys enum states */ current_state, next_state;
// synopsys state_vector current_state
reg red, yellow, green;

/*------- Sequential Logic ----*/
always@(posedge clock or negedge reset)
if (!reset)    current_state <= S0;
else  current_state <= next_state;

/* next state logic and output logic */
always@(current_state or car)
begin
red = 0; yellow = 0; green = 0;  /* defaults to prevent latches */
case (current_state) // synopsys full_case parallel_case
S0: begin
red = 1;
if (car) next_state = S1
else next_state = S0;
End
S1: begin
yellow = 1;
next_state = S2;
End
S2 : begin
green = 1;
next_state = S0;
End
default : next_state = S0;
Endcase
end

endmodule
```



FSM Verilog Notes

- Code each FSM by itself in one module.
- Separate Sequential and Combinational Logic
- Is this reset Synchronous or Asynchronous?
  - Asynchronous usually results in less logic (reset is actually synchronized when it enters the chip).
- Note use of Synthesis directives:
  - `//synopsys enum states` and `//synopsys state_vector current_state` tell Synopsys what the state vector is.
    - You can optionally use Synopsys FSM optimization procedures
  - Why can we state `//synopsys full_case parallel_case` for FSMs?
- How to we prevent accidently inferring latches?

FSM State Encoding Options

- Can either do `by hand' in Verilog source code or by reassigning states in Synopsys:
  - Binary or Sequential (minimal) encoding:

```
State 0 = 000
State 1 = 001, etc.
```

  - Gray encoding gives the minimum change in the state vector between states:

```
State 0 = 000
State 1 = 001
State 2 = 011, etc
```

    - Reduces state transition errors caused by asynchronous inputs changing during flip-flop set-up times.
    - Minimizes power consumed in state vector flip-flops

```
Synopsys: set_fsm_encoding_style gray //+ See manual
```

  - One-hot encoding assigns one flip-flop per state:

```
State 0 = 0001
State 1 = 0010
State 2 = 0100, etc
```

    - Fastest but largest design

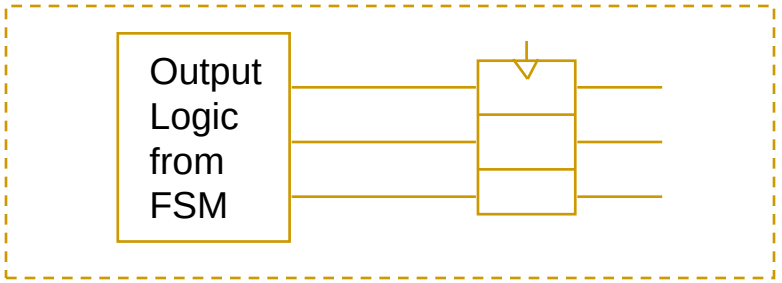
```
Synopsys:  set_fsm_encoding_style one_hot
```

  - Custom: Assign states by hand in Verilog of Synopsys

Registering FSM Outputs

- Sometimes useful to **register** the outputs of FSMs:
  - Necessary when these outputs are interfacing asynchronously with other modules or off-the-chip
    - e.g. RAS and CAS outputs for a memory interface
  - Useful if delays in output combinational logic are making it hard to meet timing requirements in the module they are connected to.
    - assumes flip-flop t_cp_Q is faster (might not be - look in library sheets)
- e.g.

```
always@(posedge clock)
  begin
    red <= int_red;
    yellow <= int_yellow;
   green <= int_green;
  end
...
case (current_state)
  S0: begin  int_red=1;
```



  - Note: changes now delayed one clock when compared with previous version

Review Questions

- What is the final Mantra?
- What types of controllers are there?
- What coding style is used for FSMs?