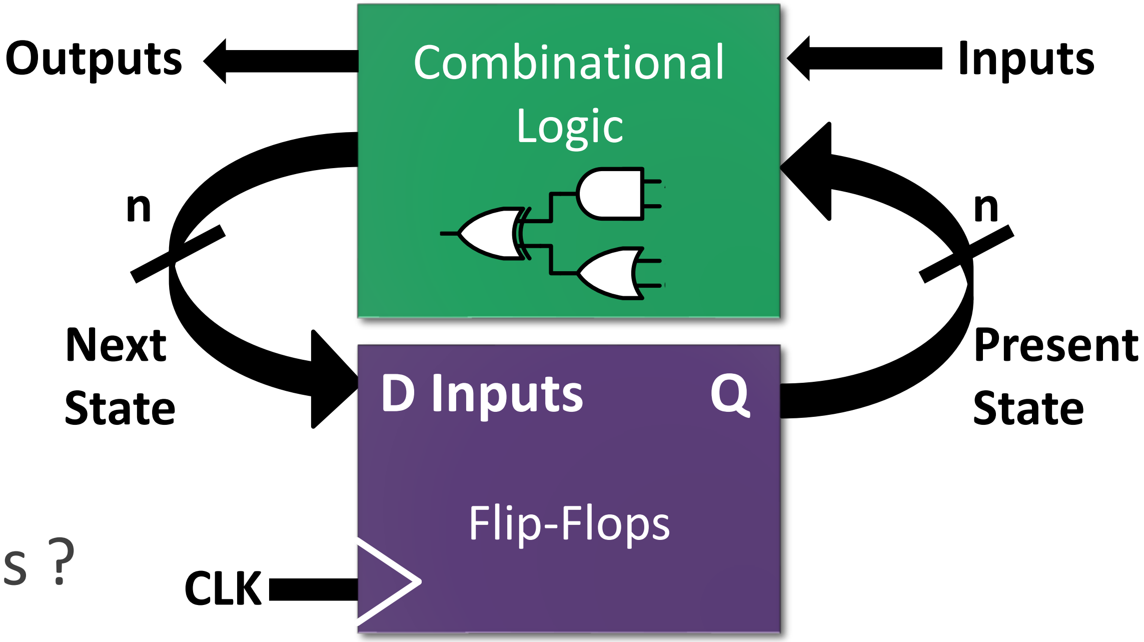
FSM : Finite State Machines

What is a FSM?

- FSM : Finite State Machine
- Synchronous Machine with “states” of operation
- At each *active clock edge*, combinational logic computes **outputs** and **next state,** as a function of **inputs and present state**



- Examples ?

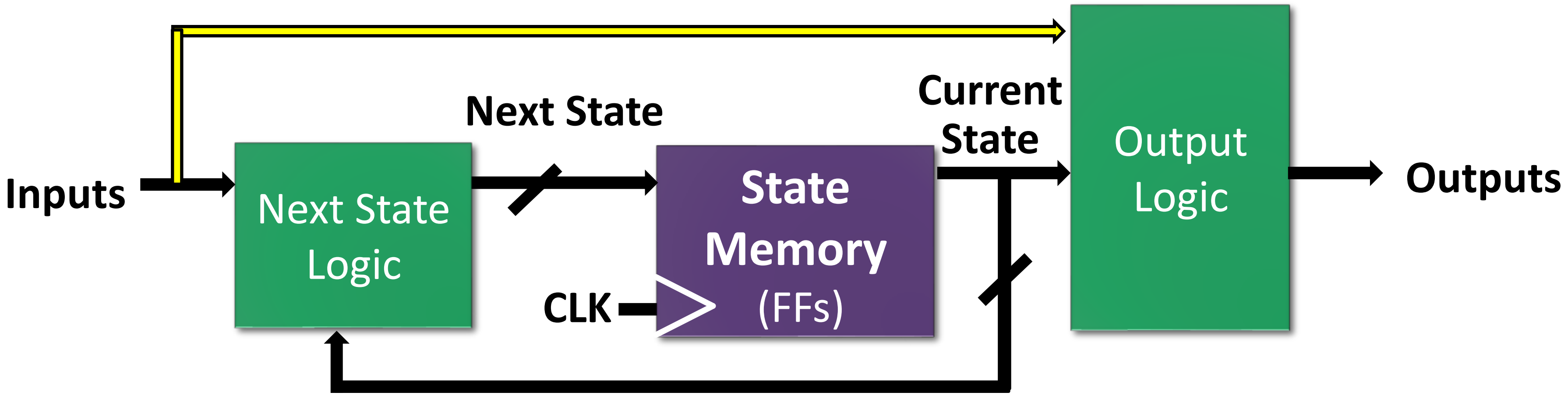
● 考试重点

- Moore：输出写在“状态里面”
- Mealy：输出写在“边（transition）上”

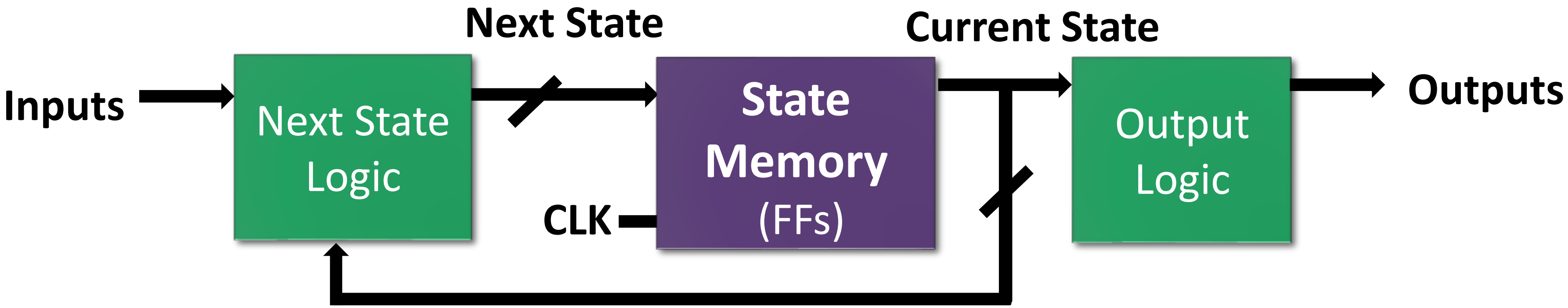
Mealy and Moore

Moore vs Mealy FSMs : Different Output Generation

○ Mealy machine: *output is a function of a **present state & inputs**.*



○ Moore machine: *output is a function of a **present state only**.*



Structure

State Memory

- set of n FFs store current state of machine; up to 2^n states.
- FFs can be J-K or D, but D FFs simpler (1 input vs. 2 inputs for J-K FFs)

Next State Logic

- combinational circuit which decides the next state of the machine based on current state and inputs:

Next state = f (inputs, current state)

Output logic

- combinational circuit which creates the output

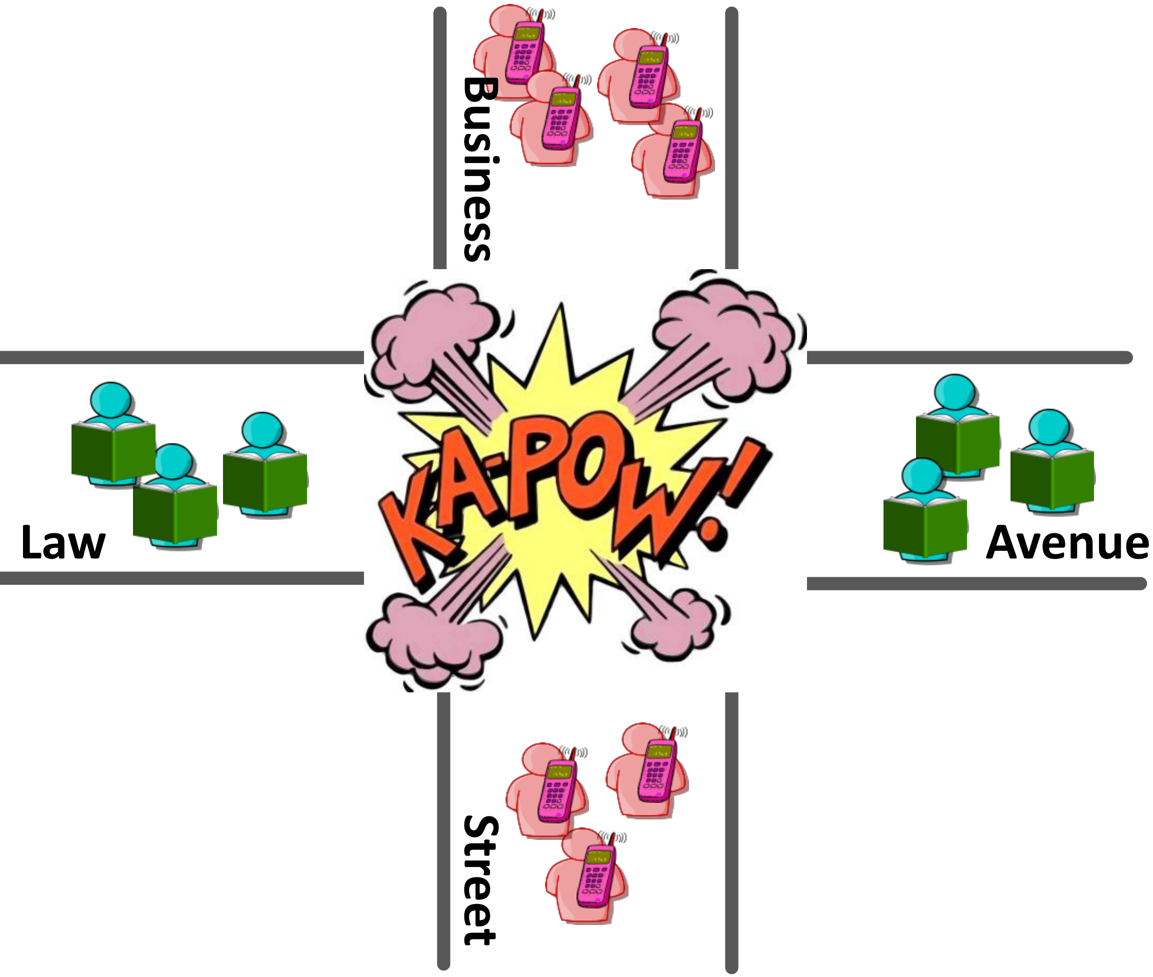
  Moore : outputs depend on current state

  outputs = g (current state)

  Mealy : outputs depend on the current state & inputs

  outputs = g (inputs, current state)

Traffic Problem…

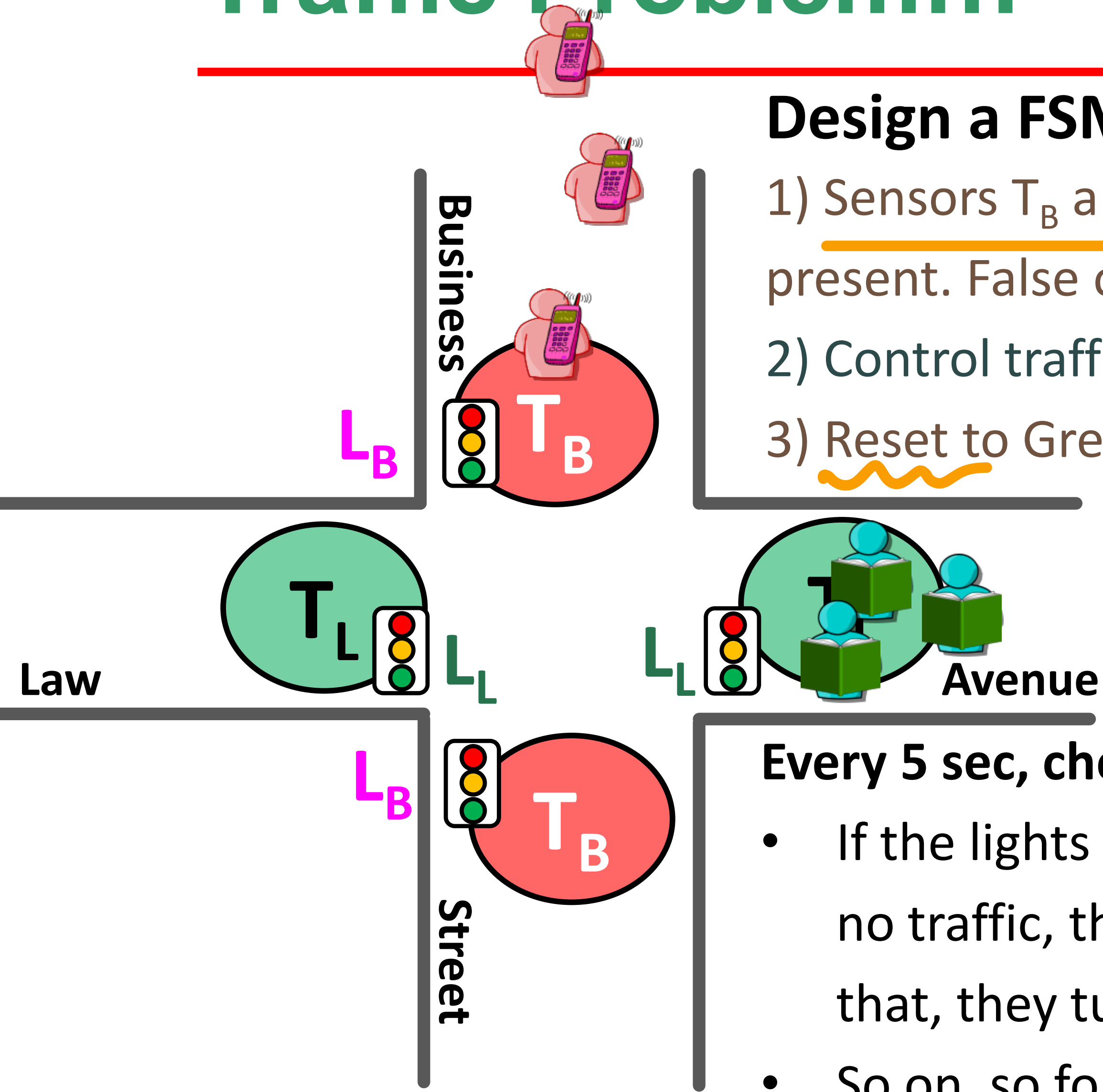


At a busy intersection on campus…

Students from the Law faculty are burying their heads in their books and are not looking where they are going.

Students from the Business faculty are occupied on their phones and aren’t looking at where they’re going either….

Traffic Problem…



Design a FSM to control the traffic lights!

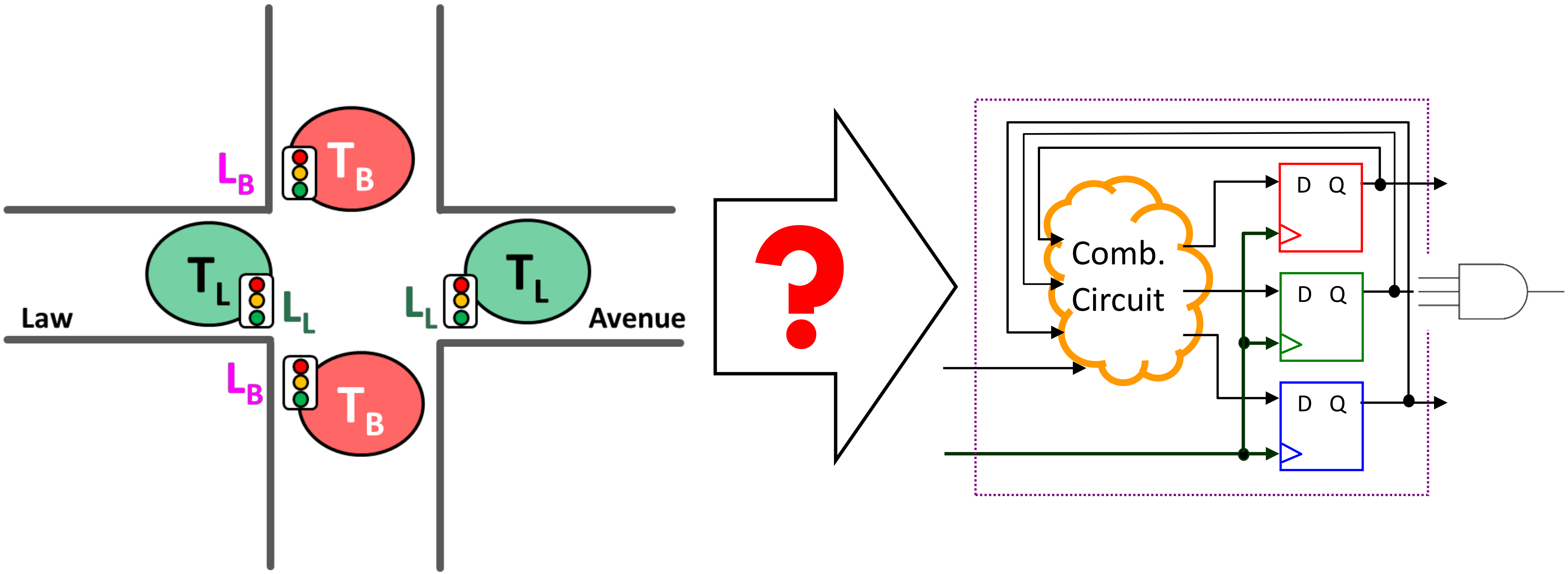
1) Sensors T_B and T_L are TRUE when students are present. False otherwise.

2) Control traffic lights L_L, L_B to be green, yellow, red.

3) Reset to Green on Law Ave and Red on Business St.

Every 5 sec, check the traffic and decide what to do!

- If the lights on Business St. are green and there's no traffic, the lights turn yellow for 5 secs. After that, they turn red and Law Ave. lights turn green.
- So on, so forth.
- If there is traffic, lights do not change.

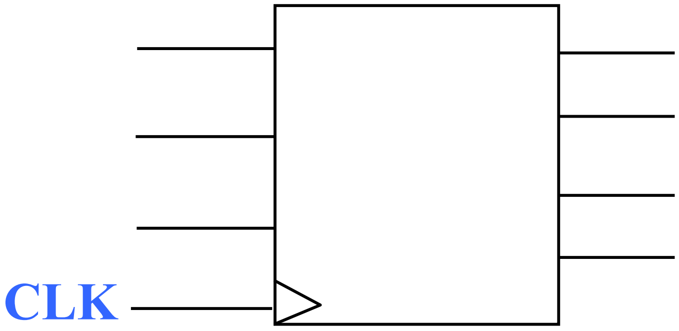


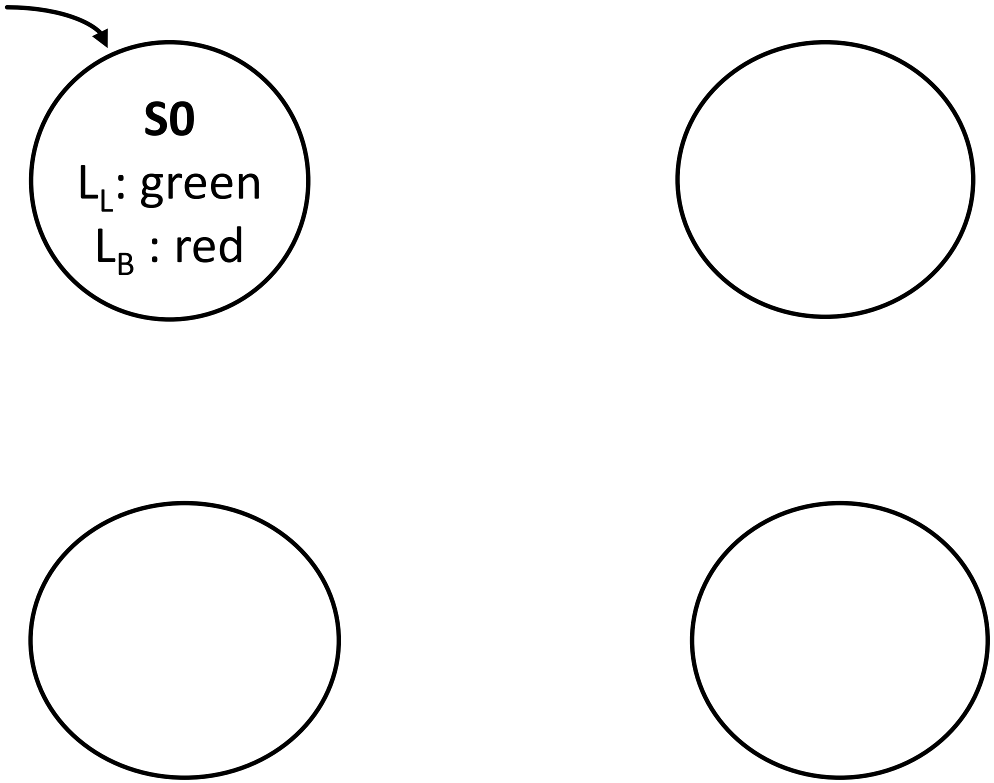
A SYSTEMATIC PROCEDURE TO GO FROM

IDEA → IMPLEMENTATION

Step 1 : State Transition Diagram

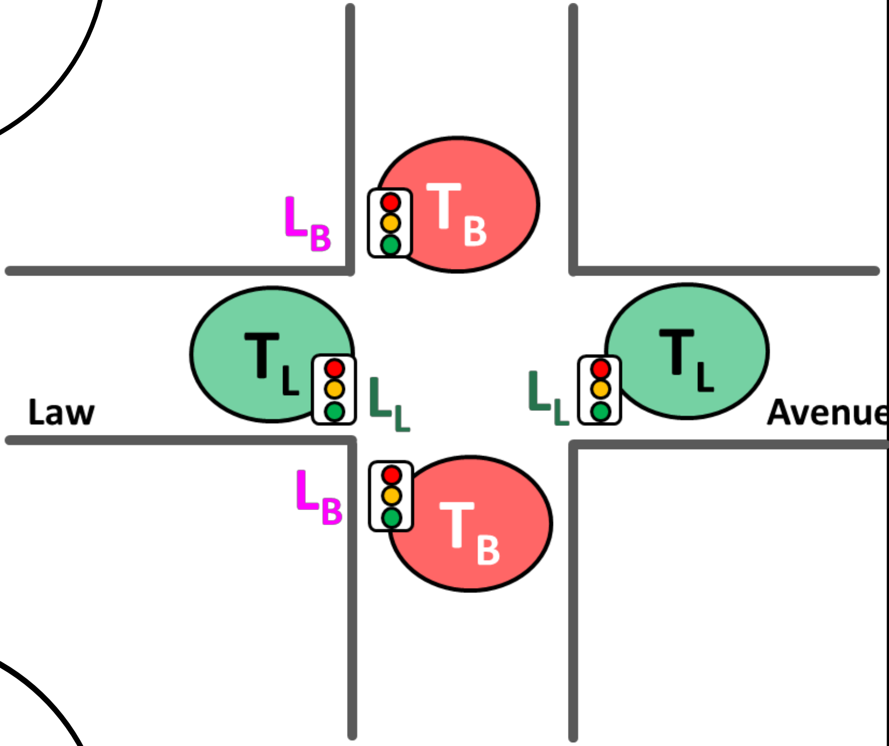
- ○ Block Diagram of Desired System
- ○ Design ***State Transition Diagram*** to represent FSM





S0
L_L: green
L_B : red

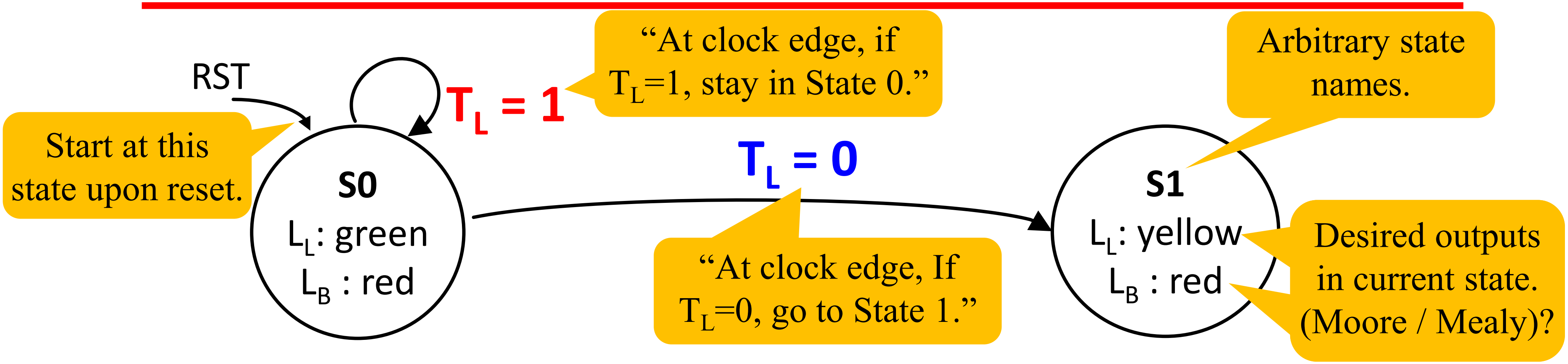
S1
L_L:yellow
L_B : red



S3
L_L: red
L_B:yellow

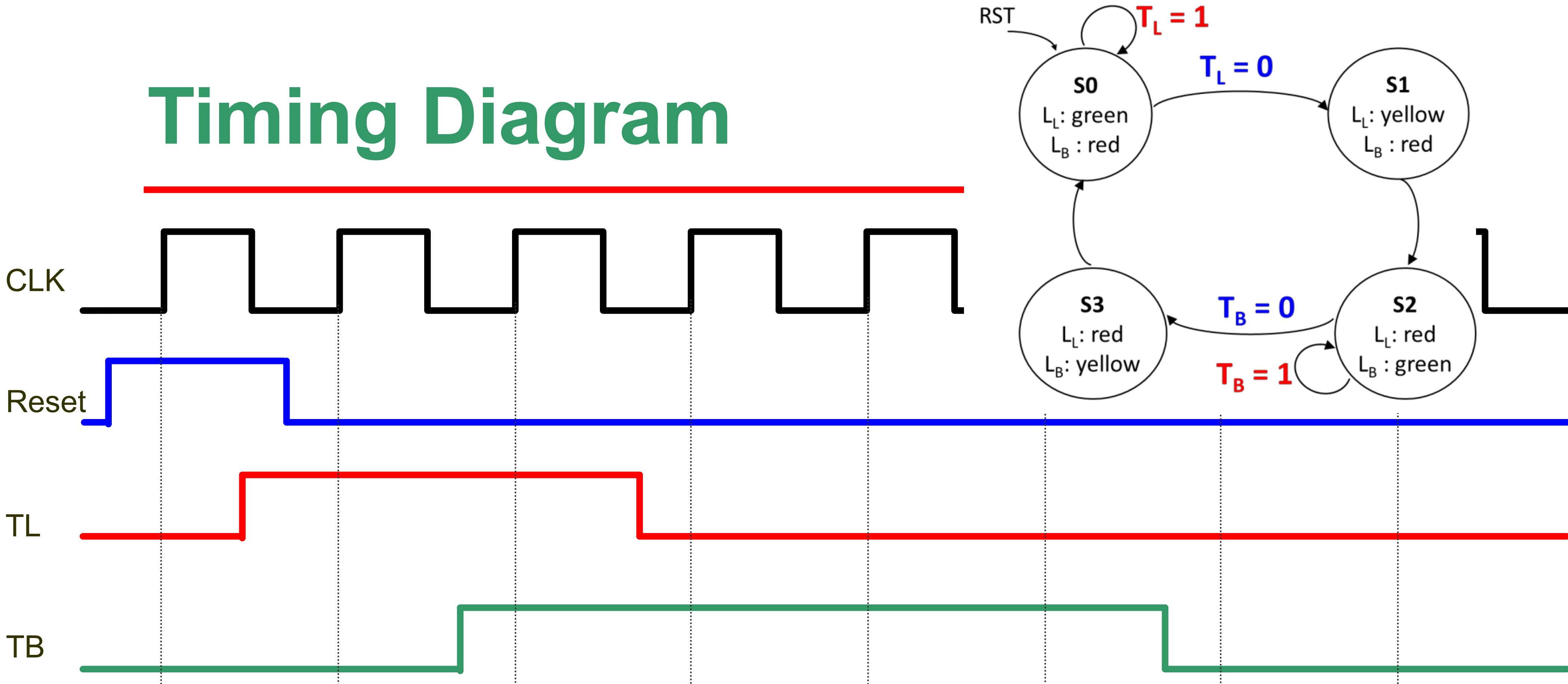
S2
L_L: red
L_B:green

Moore State Transition Diagrams


RST
Start at this state upon reset.
S0
L_L: green
L_B : red
$T_L = 1$
"At clock edge, if T_L=1, stay in State 0."
$T_L = 0$
"At clock edge, If T_L=0, go to State 1."
Arbitrary state names.
S1
L_L: yellow
L_B : red
Desired outputs in current state. (Moore / Mealy)?

- Circles represent ***states***.
  - * Each state specifies values for all outputs (Moore)
- Arcs represents ***transitions*** between states.
  - * Labels ➔ input that triggers the transition.
  - * Transitions take place on the active edge of the clock.
- Arc from outer space indicates initial state upon reset
- Within each state, for any combination of input values, there's exactly one applicable arc.

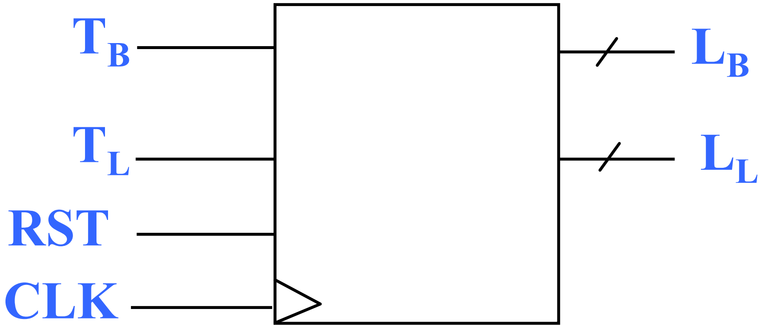
Timing Diagram

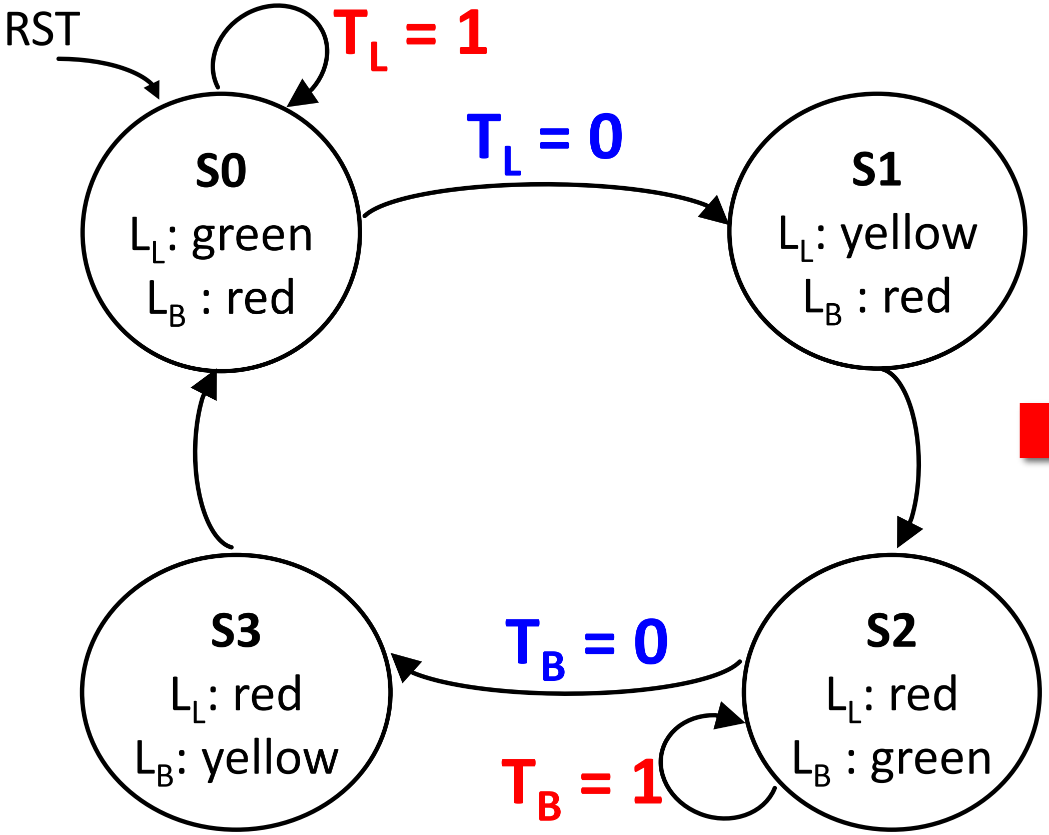


State	**S0**	S_0	S_0	S_1	S_2	S_2	S_3	S_0
LL	**Green**	G	G	Y	R	R	R	G
LB	**Red**	R	R	R	G	G	Y	R

Step 1 : State Transition Diagram

- ○ Block Diagram of Desired System
- ○ Design ***State Transition Diagram*** to represent FSM





RST

$T_L = 1$

$T_L = 0$

S0
L_L: green
L_B : red

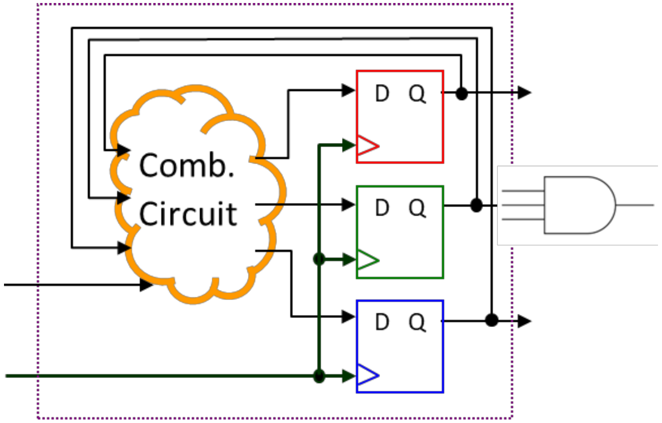
S1
L_L: yellow
L_B : red

S2
L_L: red
L_B : green

S3
L_L: red
L_B: yellow

$T_B = 0$

$T_B = 1$



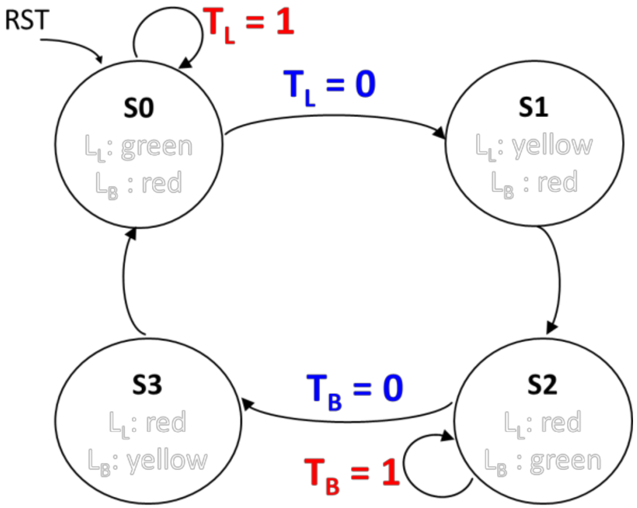
Step 2 : Next State Table

1. From STD, find the number of states 4
2. Number of bits / FFs required = ? 2 to go through these states 2^N
3. State Assignment
   1. Sequential
   2. One-hot (100,010,001)
   3. Gray
   4. Johnson
   5. Output Encoded

State	S_1S_0
S0	00
S1	01
S2	10
S3	11

4. Next State Table

Current State	Inputs		Next State
S	T_L	T_B	S+
S0	0	X	S1
S0	1	X	S0
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0



Step 2 : Next State Table

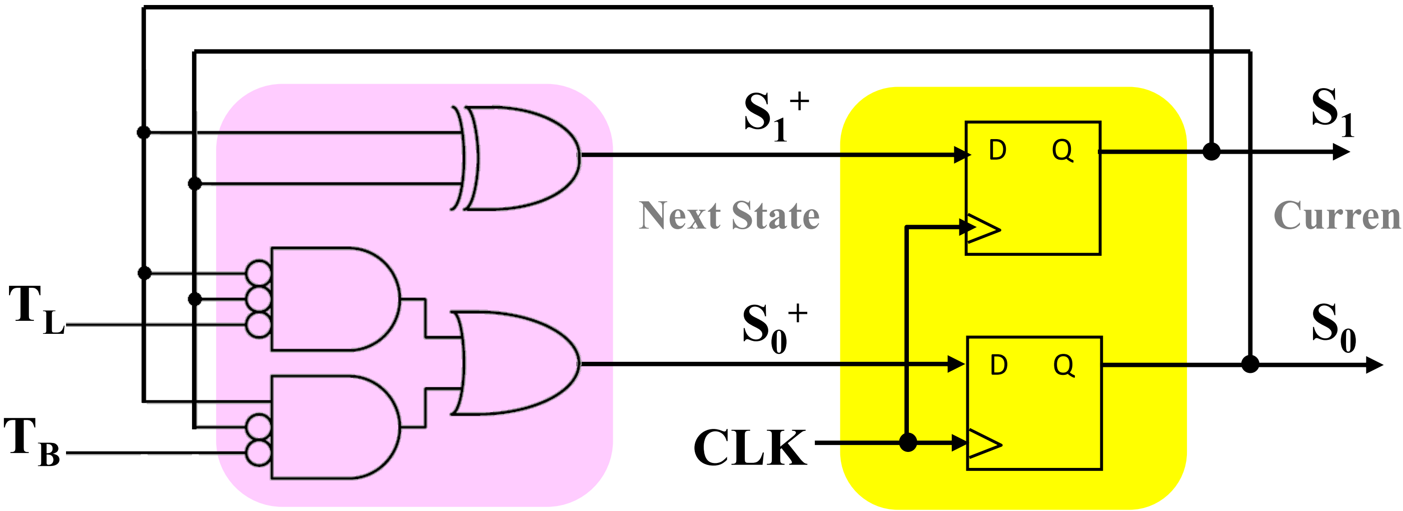
5. State Generator Circuit

Current State		Inputs		Next State	
S_1	S_0	T_L	T_B	S_1+	S_0+
0	0	**0**	X	0	1
0	0	**1**	X	0	0
0	1	X	X	1	0
1	0	X	**0**	1	1
1	0	X	**1**	1	0
1	1	X	X	0	0

$$S_1^+ = \overline{S_1} S_0 + S_1 \overline{S_0}\,\overline{T_B} + S_1 \overline{S_0} T_B$$

$$= S_1 \oplus S_0$$

$$S_0^+ = \overline{S_1}\,\overline{S_0}\,\overline{T_L} + S_1 \overline{S_0}\,\overline{T_B}$$



Step 3 : Output Logic

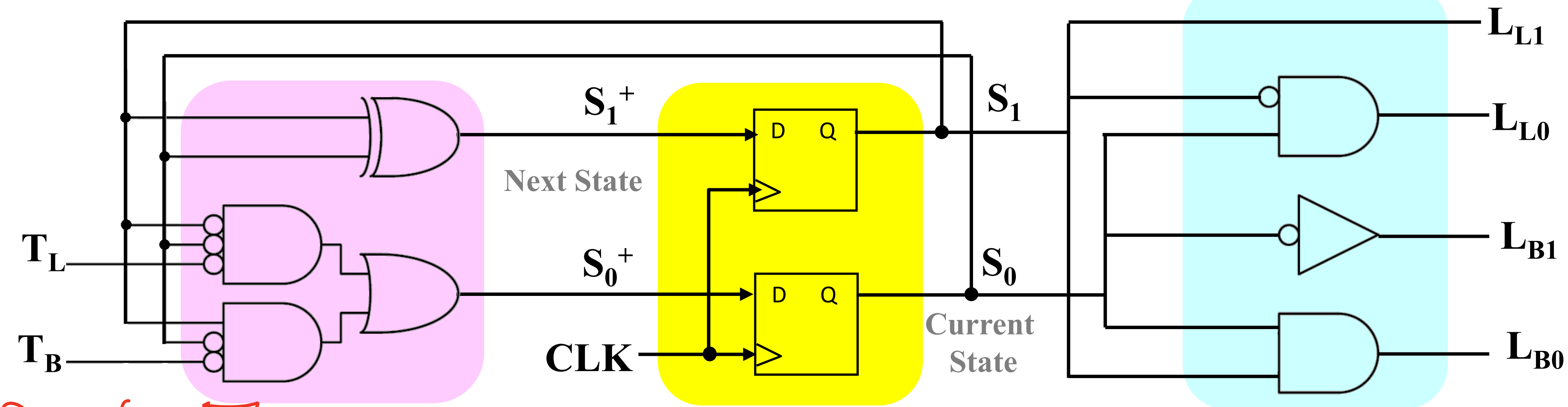
Output	L_1L_0
Green	00
Yellow	01
Red	10

1. Output Truth Table

Current State		Outputs			
S_1	S_0	L_{L1}	L_{L0}	L_{B1}	L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

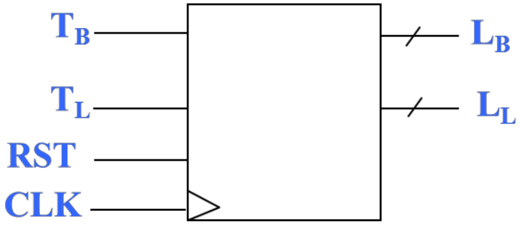
$$L_{L1} = S_1 \qquad L_{L0} = \overline{S_1}S_0$$

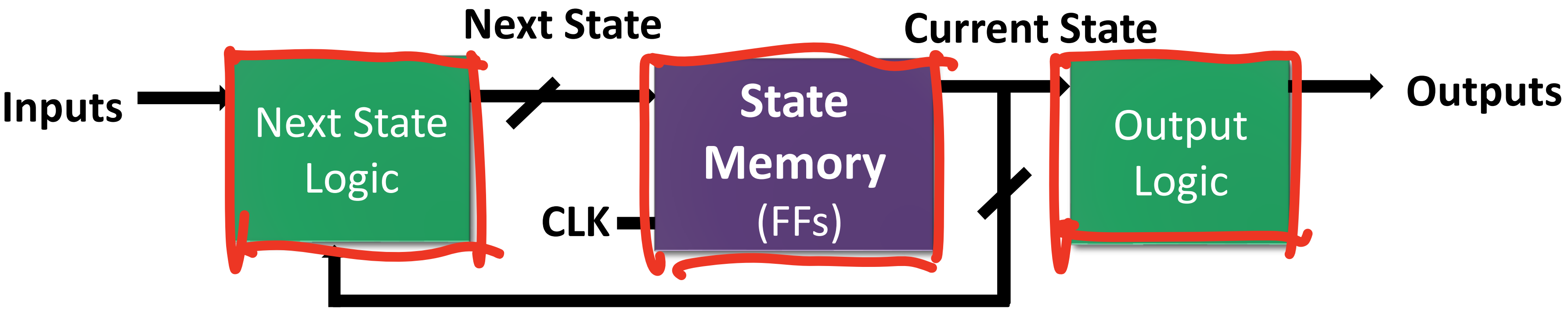
$$L_{B1} = \overline{S_1} \qquad L_{B0} = S_1S_0$$

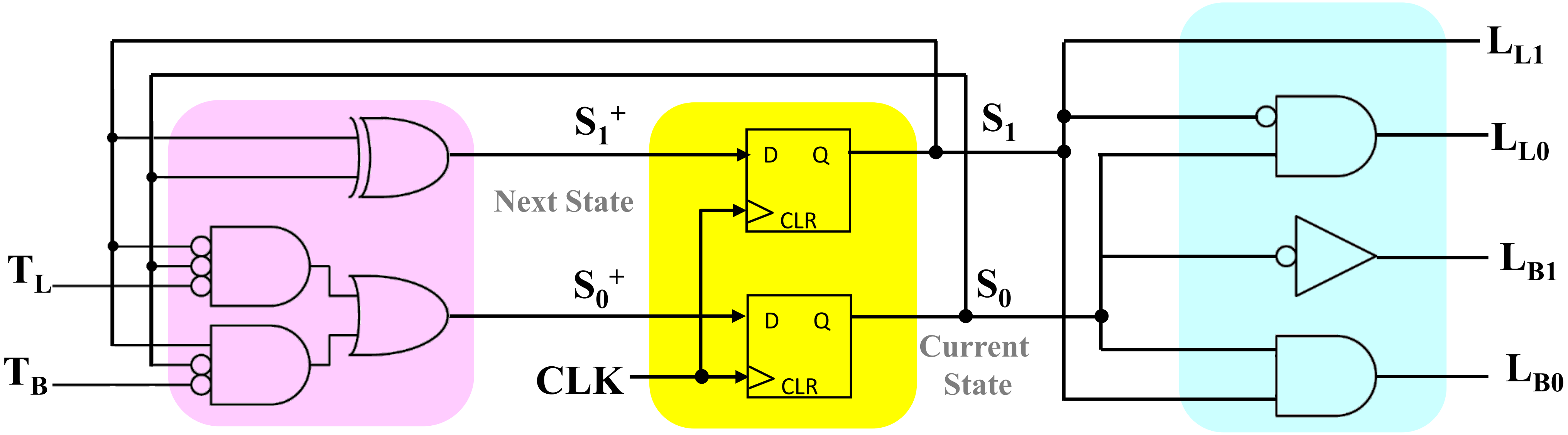


Reset

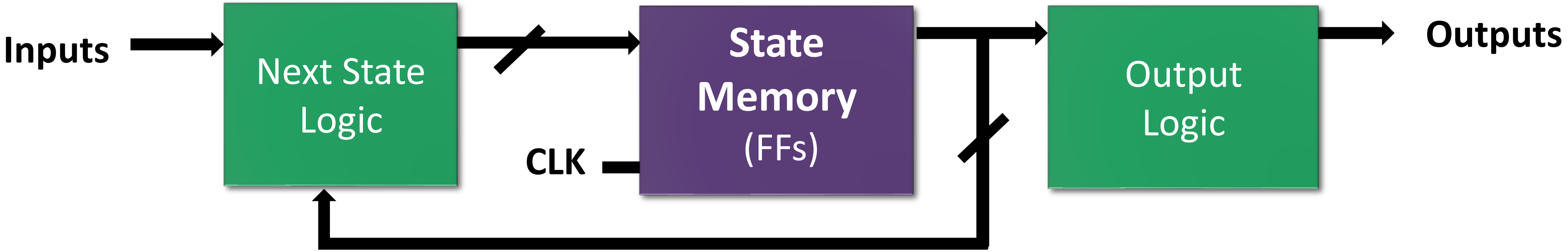
Ta-Dah!


T_B
T_L
RST
CLK
L_B
L_L


Inputs
Next State Logic
Next State
CLK
State Memory (FFs)
Current State
Output Logic
Outputs


T_L
T_B
S_1^+
Next State
S_0^+
CLK
D
Q
CLR
D
Q
CLR
S_1
S_0
Current State
L_{L1}
L_{L0}
L_{B1}
L_{B0}

Verilog~! – Code Structure



Combinational Circuit

```
always @ (*)
```

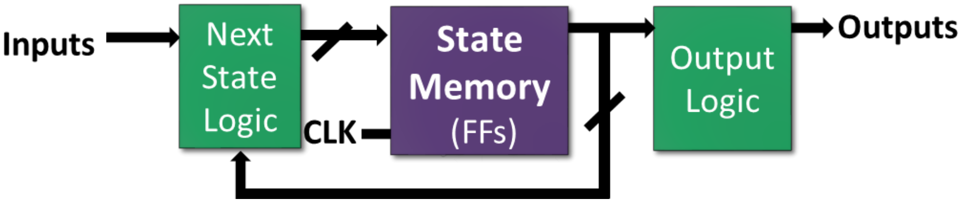
Sequential Circuit

```
always @
 (posedge clk)
```

Combinational Circuit

```
assign y =
```

FSM in Verilog



```
module fsm(input clk, … , output … … );
reg __ state, nextstate;
```

1 bit

2个状态

```
parameter S0 = 2’b00;
parameter S1 = 2’b01;
```

parameter is used to define constants within a module, improving code readability.

```
always @ (*)
    case (state)
        S0  : nextstate = S1;
        S1  : nextstate = S0;
    endcase
```

Next State Combinational Logic :
(*) code is triggered whenever any input changes ➔ combinational logic.
case represents next state table.

状态转移表

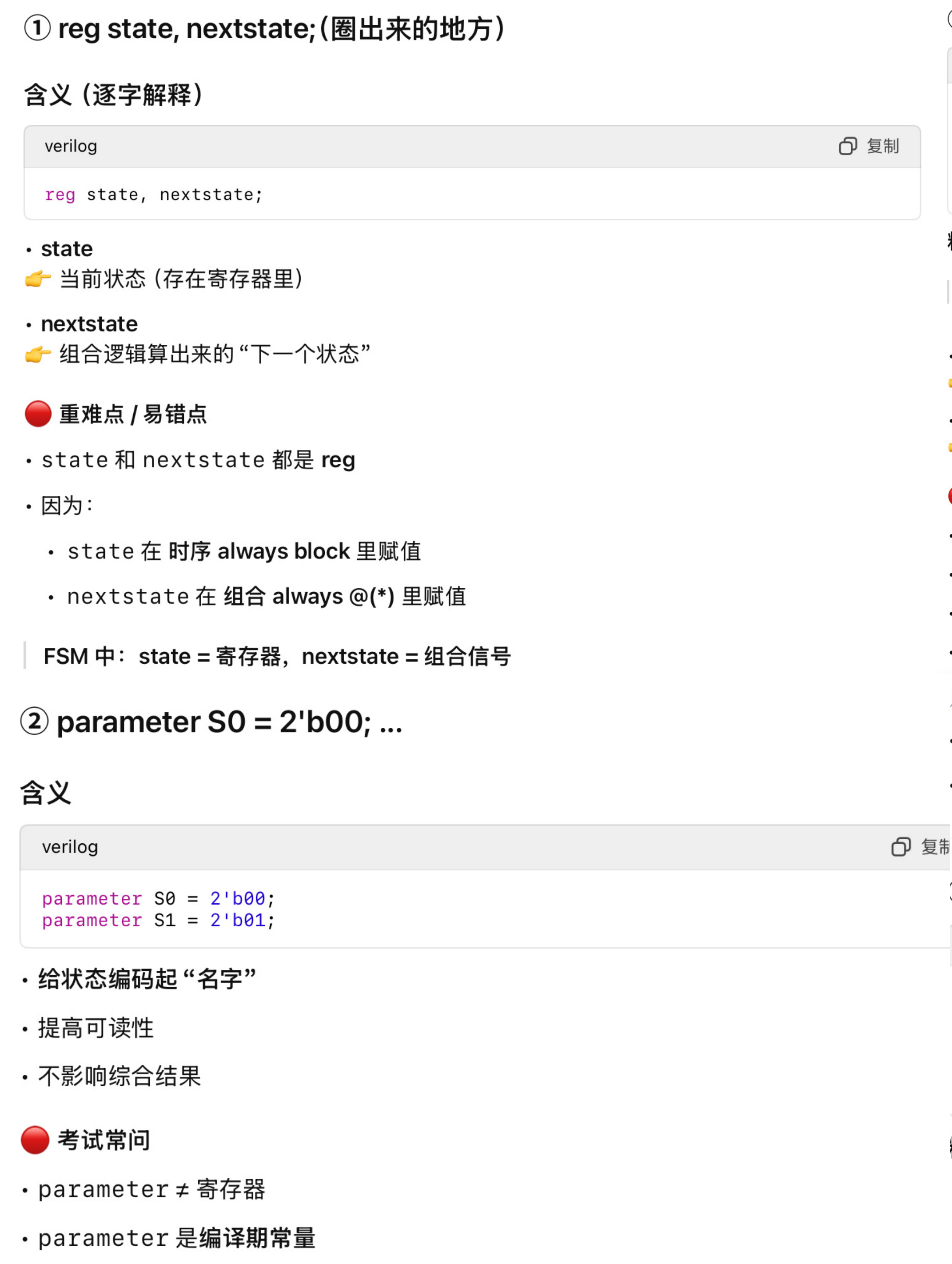
```
always @ (posedge clk, posedge reset)
    if (reset)  state <= S0;
    else        state <= nextstate;
```

Sequential Logic :
Use **<=** to infer flip-flops.
Is this Sync or Async reset? ✓

```
assign y = ( state == S0 );

endmodule
```

Output Logic : Use assign to infer combinational logic.

Equality Comparison :
a == b evaluates to 1 if a equals b.



① reg state, nextstate;（圈出来的地方）

含义（逐字解释）

```verilog
reg state, nextstate;
```

- **state**
👉 当前状态（存在寄存器里）
- **nextstate**
👉 组合逻辑算出来的“下一个状态”

🔴 重难点 / 易错点

- state 和 nextstate 都是 **reg**
- 因为：
  - state 在 **时序 always block** 里赋值
  - nextstate 在 **组合 always @(*)** 里赋值

> **FSM 中：state = 寄存器，nextstate = 组合信号**

② parameter S0 = 2'b00; ...

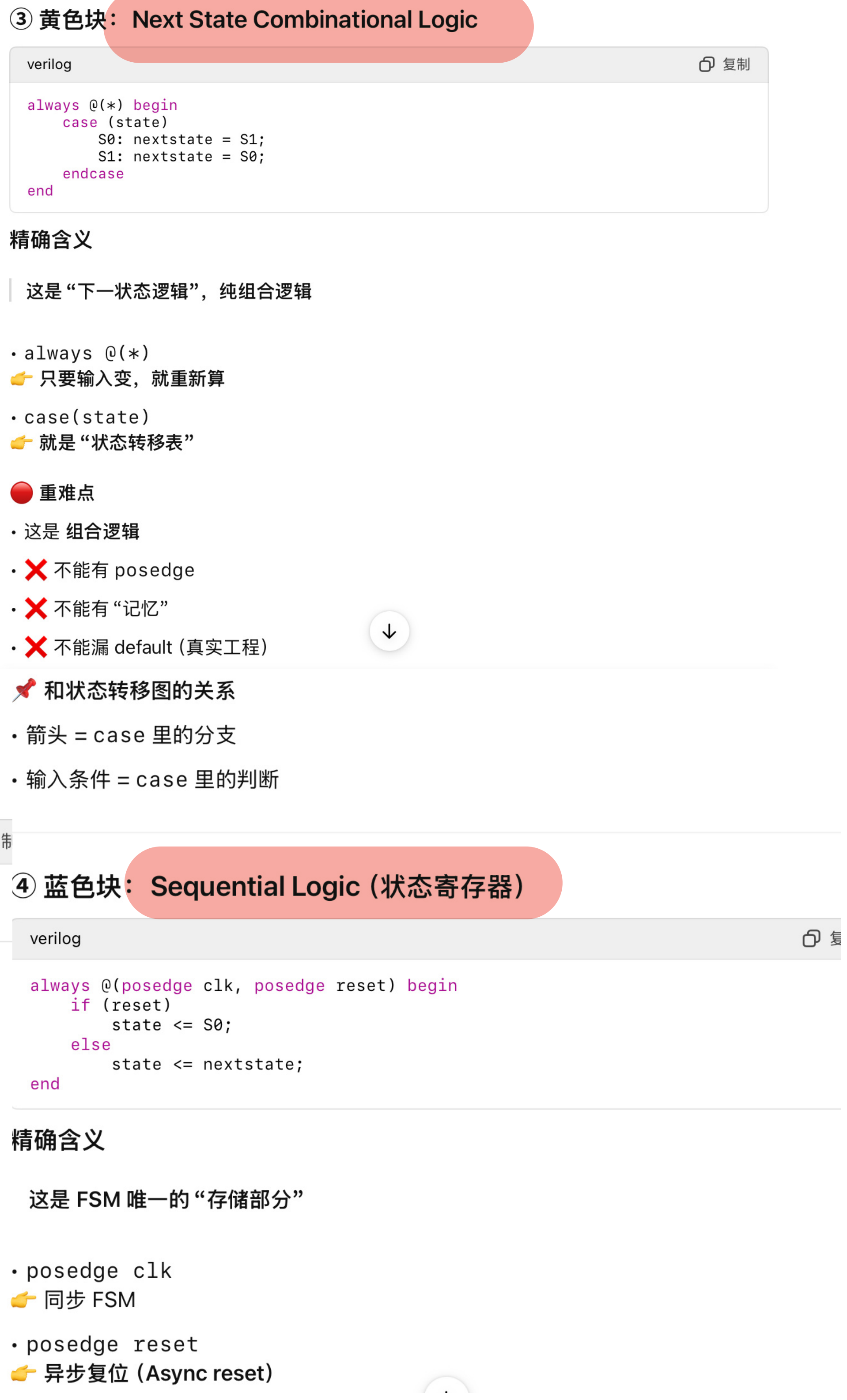
含义

```verilog
parameter S0 = 2'b00;
parameter S1 = 2'b01;
```

- 给状态编码起“名字”
- 提高可读性
- 不影响综合结果

🔴 考试常问

- parameter ≠ 寄存器
- parameter 是编译期常量



③ 黄色块：Next State Combinational Logic

```verilog
always @(*) begin
    case (state)
        S0: nextstate = S1;
        S1: nextstate = S0;
    endcase
end
```

精确含义

> **这是“下一状态逻辑”，纯组合逻辑**

- always @(*)
👉 **只要输入变，就重新算**
- case(state)
👉 **就是“状态转移表”**

🔴 重难点

- 这是 **组合逻辑**
- ❌ 不能有 posedge
- ❌ 不能有“记忆”
- ❌ 不能漏 default（真实工程）

📌 和状态转移图的关系

- 箭头 = case 里的分支
- 输入条件 = case 里的判断

④ 蓝色块：Sequential Logic（状态寄存器）

```verilog
always @(posedge clk, posedge reset) begin
    if (reset)
        state <= S0;
    else
        state <= nextstate;
end
```

精确含义

> **这是 FSM 唯一的“存储部分”**

- posedge clk
👉 同步 FSM
- posedge reset
👉 **异步复位（Async reset）**

🔴 **考试必考点**

> ❓ *Is this Sync or Async reset?*
> ✔️ **Async reset**

📌 判断规则（直接背）：

- reset 在 sensitivity list 里 → **异步**
- reset 在 if、但不在 list 里 → **同步**

⑤ 粉色块：Output Logic（Moore）

```verilog
assign y = (state == S0);
```

精确含义

> **输出只由当前状态决定**

🔴 **这是 Moore 的铁证**

- 输出：
  - ❌ 不看输入
  - ✅ 只看 `state`

📌 如果是 Mealy：

- 输出逻辑里一定会出现 **输入信号**

三、这页在“暗考”你什么？

🔥 考点 1：FSM 的三段式结构（必背）

部分	写法
Next State Logic	`always @(*)`
State Register	`always @(posedge clk …)`
Output Logic	`assign` / `always @(*)`

🔥 考点 2：RTL 思维

这页就是教你什么叫 **RTL**：

> **寄存器 + 数据如何在时钟下流动**

🔥 考点 3：为什么不用一个 always 写完？

因为：

- 会混合组合 + 时序
- 容易 latch
- **不是标准 RTL FSM**

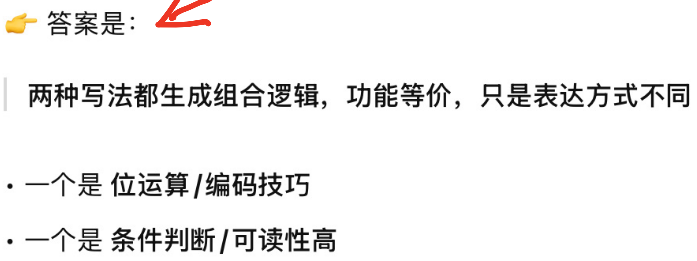
四、一句话终极总结（考试用）

> **FSM 的 Verilog 标准模板 =**
> **一个 always @(*) 算 nextstate，**
> **一个 always @(posedge clk) 存 state，**
> **一个 assign/always 生成输出。**

FSM Traffic Controller in Verilog

```
module traffic (input clk, reset,
TL, TB, output [1:0] LL, LB);

reg [1:0] state, nextstate;

parameter S0 = 2’b00, S1 = 2’b01,
S2 = 2’b10, S3 = 2’b11;

parameter green = 2’b00,
yellow= 2’b01, red= 2’b10;

always @ (*) begin
  case (state)
    S0: nextstate = TL ? S0 : S1;
    S1: nextstate = S2;
    S2: nextstate = TB ? S2 : S3;
    S3: nextstate = S0;
  endcase
end
```

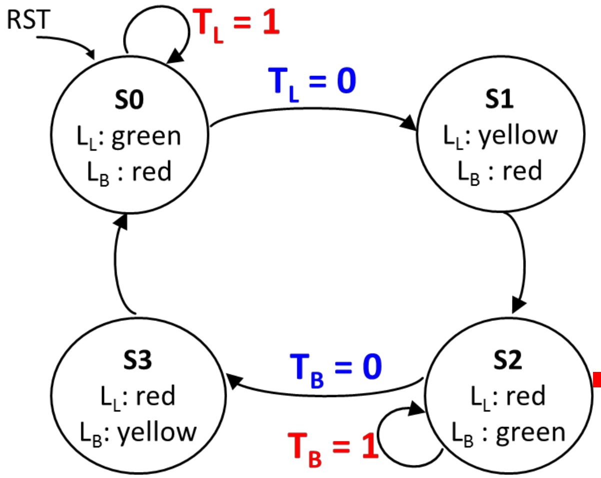
```
always@(posedge clk, posedge reset)
begin
    if (reset) state <= S0;
    else state <= nextstate;
end

//What’s the diff between these
//2 ways of coding output logic ?
assign LL
= {state[1], ~state[1] & state[0]};

assign LB =
(state== S0 || state== S1) ? red :
( (state == S2) ? green : yellow );

endmodule
```



👉 答案是：

> 两种写法都生成组合逻辑，功能等价，只是表达方式不同

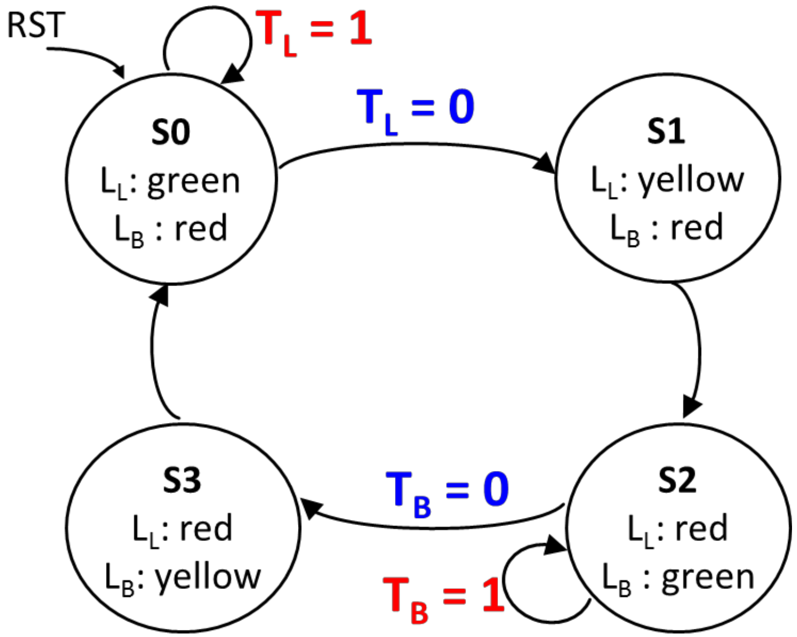
- 一个是 位运算 / 编码技巧
- 一个是 条件判断 / 可读性高

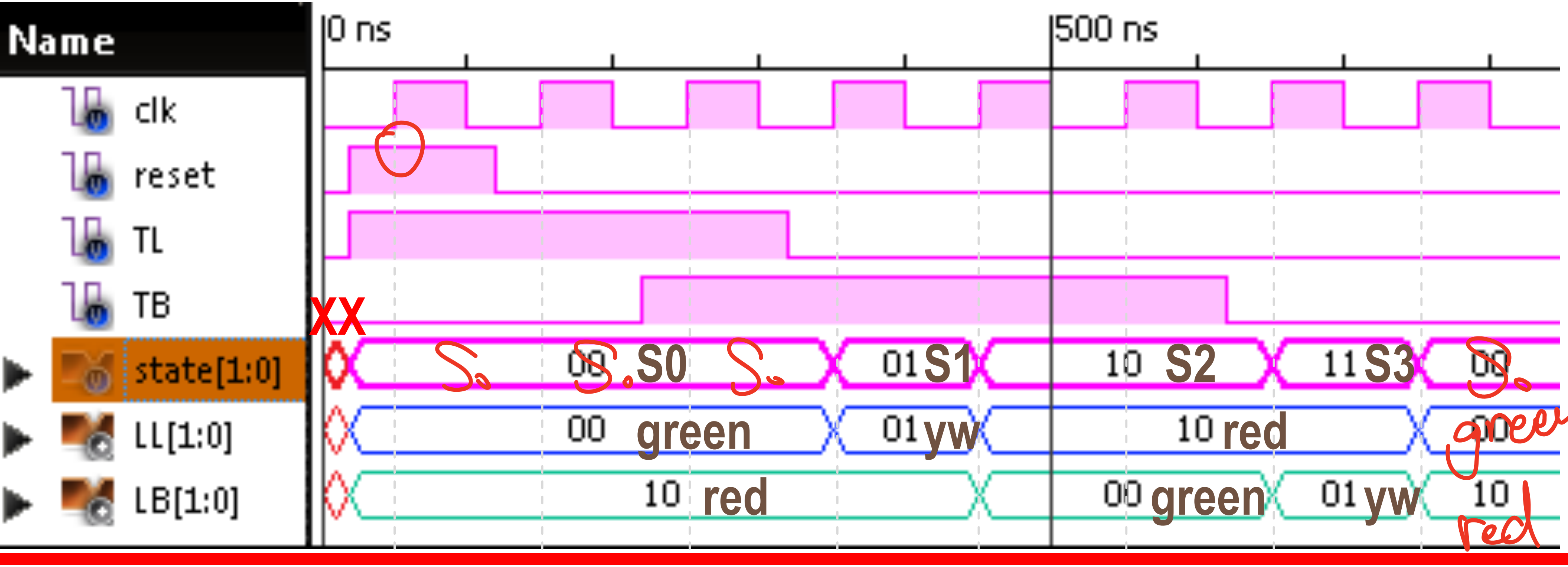


Traffic Controller

- Check the waveforms in the timing diagram below…

Are they correct?

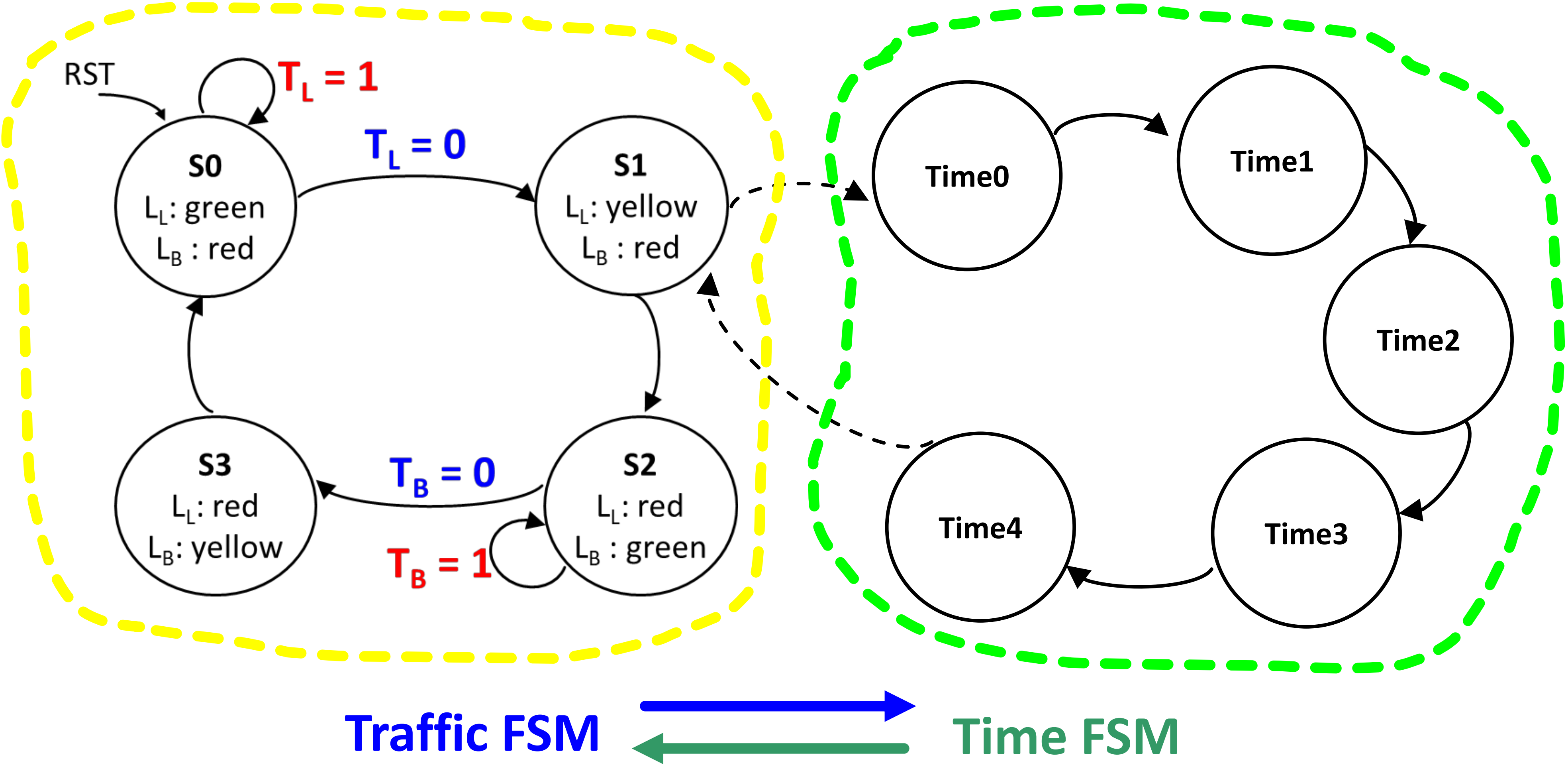




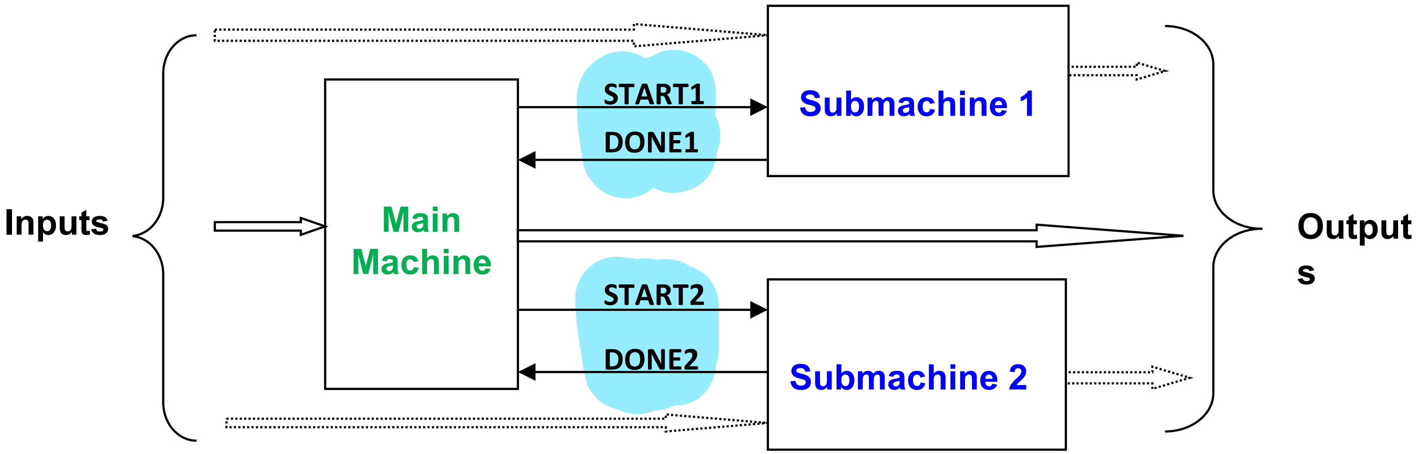
Modular Design of FSMs

What if we want to stay in state S1 for five clock cycles?

Designing complex FSMs is often easier if they can be broken down into simpler FSMs that interact.



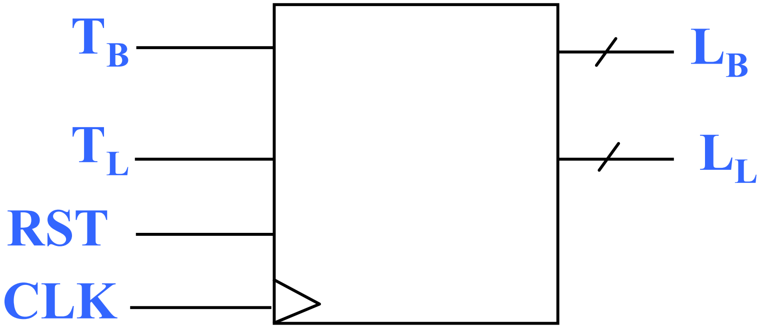
Modular Design of State Machines

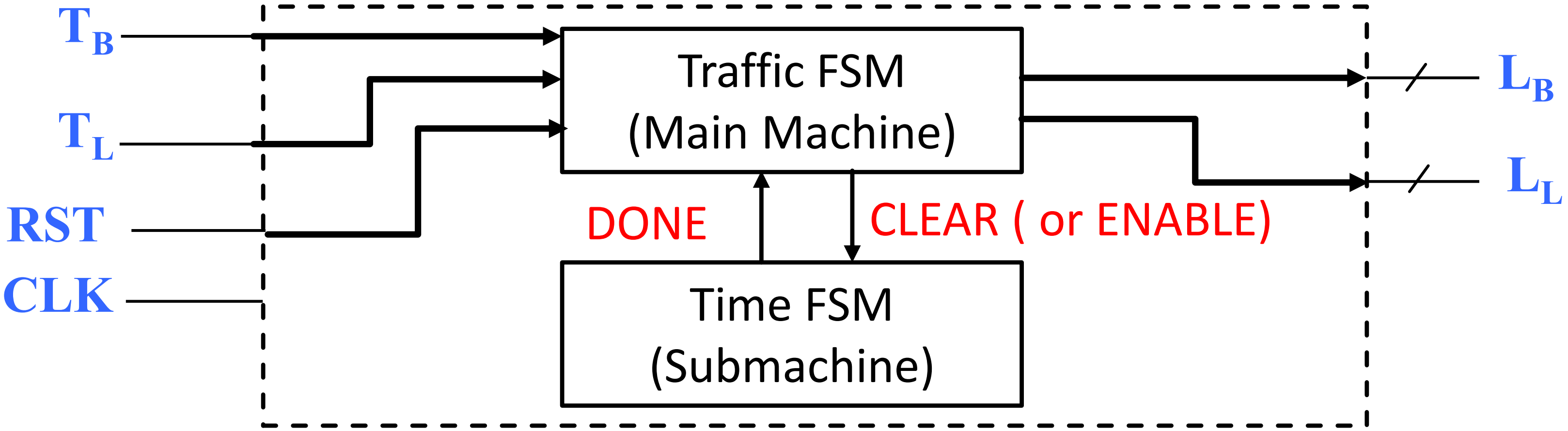


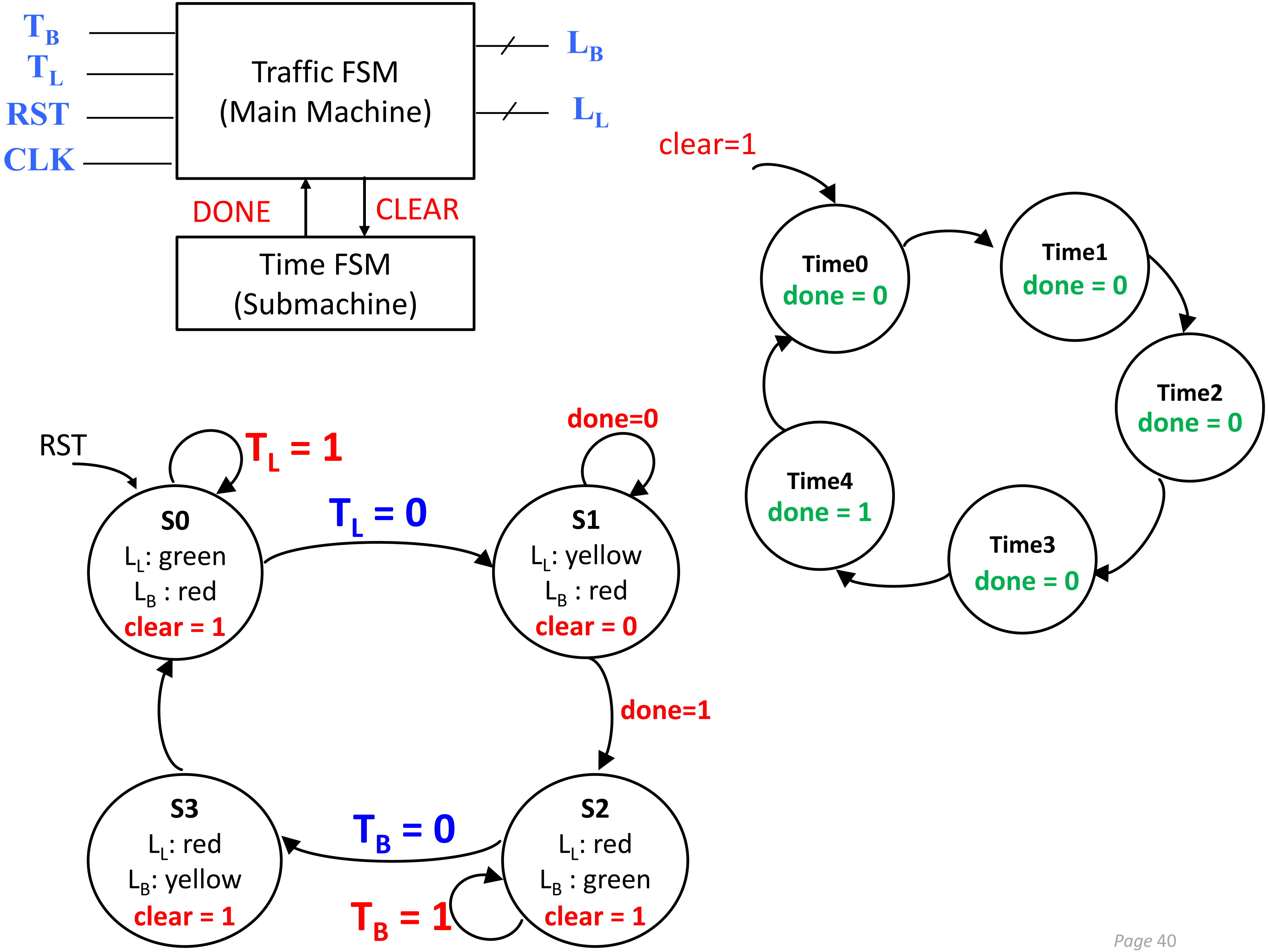
- **Main machine :** executes main algorithm, controls the submachines & get the job done. Commands & gets feedback signals from submachines.
- **Submachines** respond to external inputs & commands from main machine. Can give outputs as well as feedback to the main machine.
- Common examples of submachines are ***counters***, ***shift registers***, etc.
- Sometimes the main machine is called the *controller* and the **submachines** are called *controlled circuit elements* or ***architectural elements***.
- Trick here is to **modularize** appropriately, and pick suitable components for the submachines that simplify the design problem.

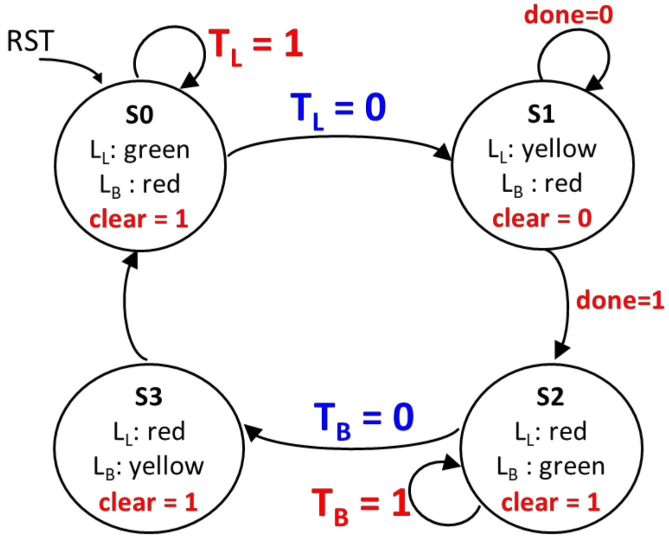
Modularizing…

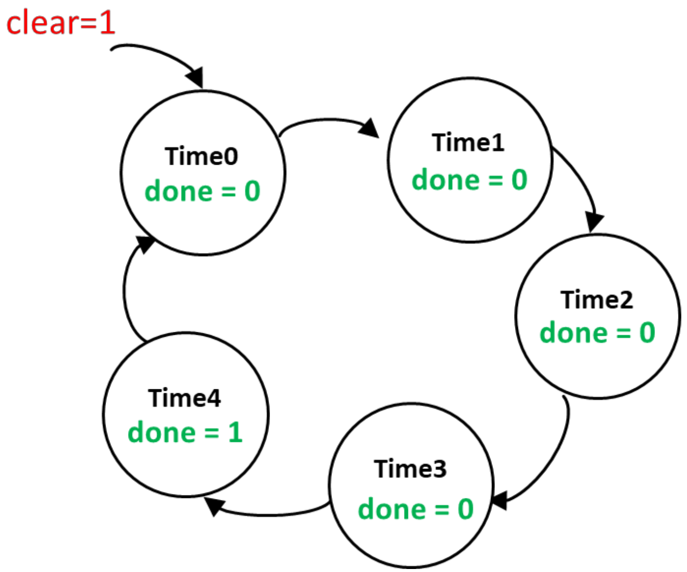
What is a natural submachine we can use?



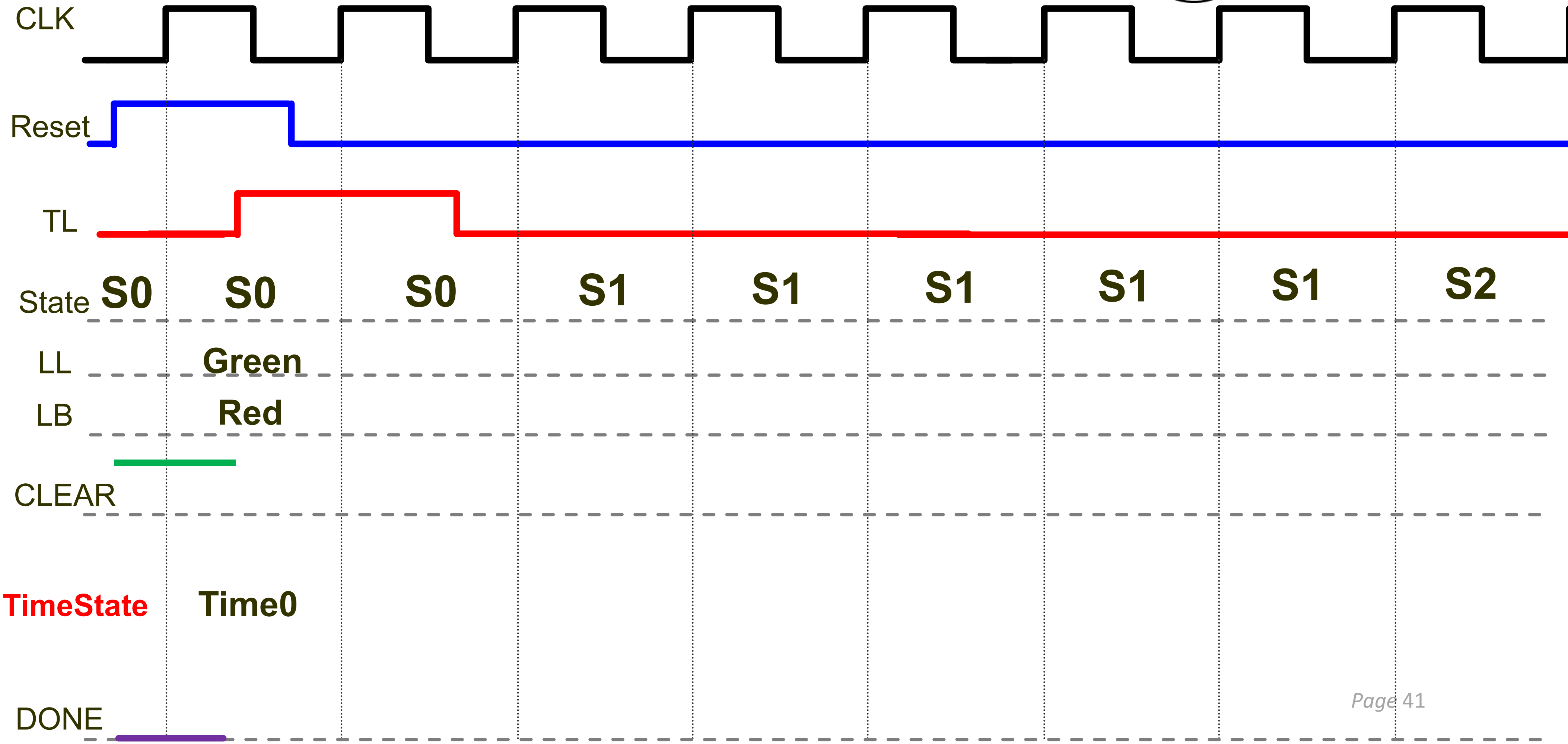



T_B
T_L
RST
CLK
Traffic FSM
(Main Machine)
L_B
L_L
DONE
CLEAR
Time FSM
(Submachine)
clear=1
Time0
done = 0
Time1
done = 0
Time2
done = 0
Time3
done = 0
Time4
done = 1
RST
$T_L = 1$
$T_L = 0$
done=0
S0
L_L: green
L_B : red
clear = 1
S1
L_L: yellow
L_B : red
clear = 0
done=1
S3
L_L: red
L_B: yellow
clear = 1
$T_B = 0$
S2
L_L: red
L_B : green
clear = 1
$T_B = 1$

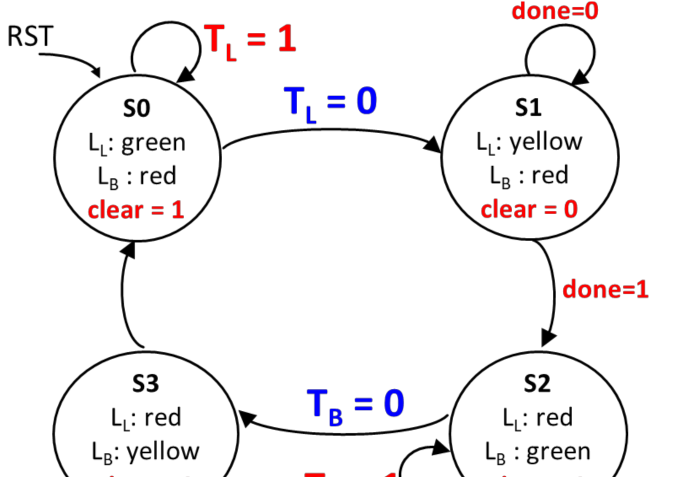

RST
$T_L = 1$
$T_L = 0$
done=0
S0
L_L: green
L_B : red
clear = 1
S1
L_L: yellow
L_B : red
clear = 0
done=1
S3
L_L: red
L_B: yellow
clear = 1
$T_B = 0$
S2
L_L: red
L_B : green
clear = 1
$T_B = 1$


clear=1
Time0
done = 0
Time1
done = 0
Time2
done = 0
Time3
done = 0
Time4
done = 1

TB = 0


CLK
Reset
TL
State
S0
S0
S0
S1
S1
S1
S1
S1
S2
LL
Green
LB
Red
CLEAR
TimeState
Time0
DONE

FSM with Submachine



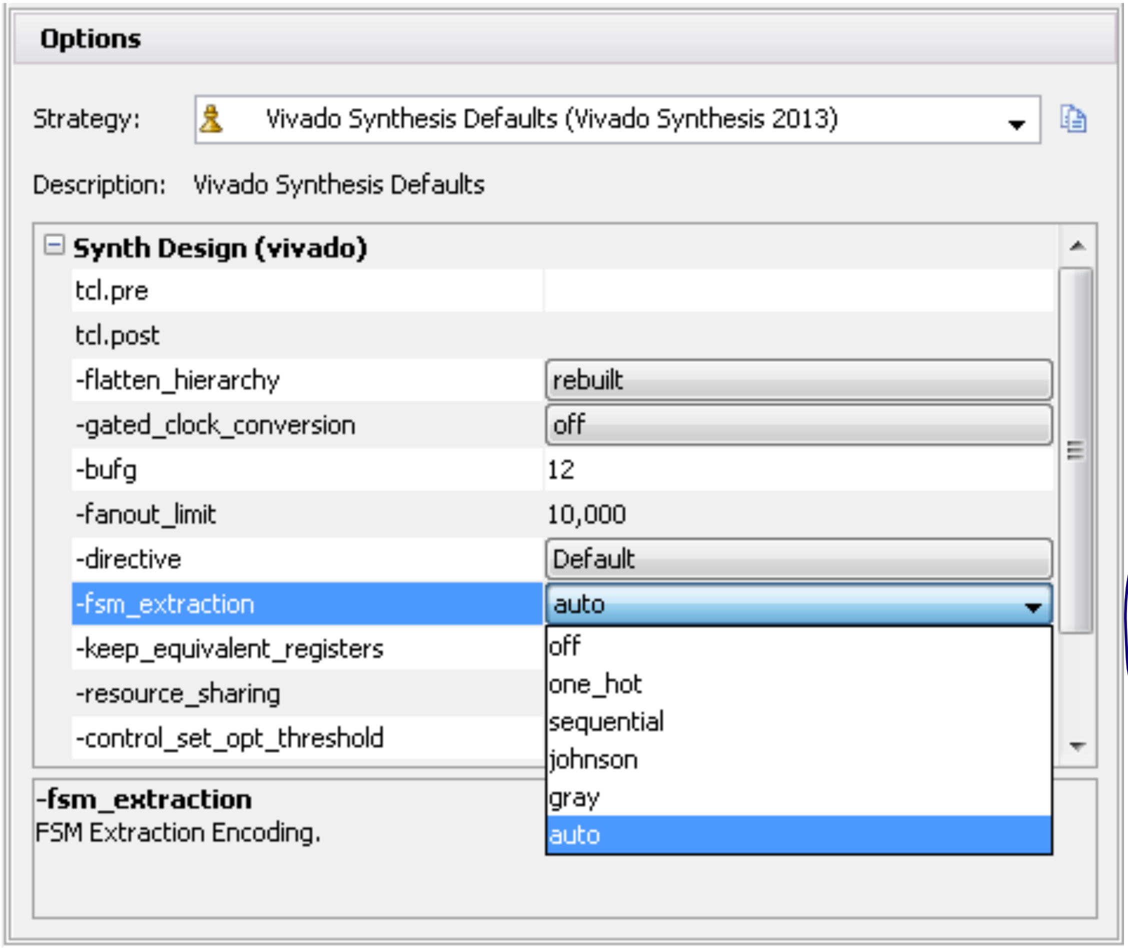
```
module traffic (input clk, reset,
TL, TB, output [1:0] LL, LB);

reg [1:0] state, nextstate;
wire clear, done;

parameter S0 = 2’b00, S1 = 2’b01,
S2 = 2’b10, S3 = 2’b11;

parameter green = 2’b00,
yellow= 2’b01, red= 2’b10;

always @ (*) begin
  case (state)
    S0: nextstate = TL ? S0 : S1;
    S1: nextstate = done? S2 : S1;
    S2: nextstate = TB ? S2 : S3;
    S3: nextstate = S0;
  endcase
end

always@(posedge clk, posedge
reset) begin
    if (reset) state <= S0;
    else state <= nextstate;
end
```

```
assign LL
= {state[1], ~state[1] & state[0]};

assign LB =
(state== S0 || state== S1) ? red :
( (state == S2) ? green : yellow );
-------------------------------------------
assign clear = (state != S1);

reg [2:0] count = 3’b000;

always @(posedge clk) begin
   if (clear) count <= 0;
   else begin
       count <= (count == 3’b100) ? 0 : count + 1;
     end
   end
```

0~4

```
assign done = (count == 3’b100);

endmodule
```

Vivado : fsm_extraction



Options

Strategy: Vivado Synthesis Defaults (Vivado Synthesis 2013)

Description: Vivado Synthesis Defaults

Synth Design (vivado)	
tcl.pre	
tcl.post	
-flatten_hierarchy	rebuilt
-gated_clock_conversion	off
-bufg	12
-fanout_limit	10,000
-directive	Default
-fsm_extraction	auto
-keep_equivalent_registers	
-resource_sharing	
-control_set_opt_threshold	

-fsm_extraction
FSM Extraction Encoding.

- INFERENCE of FSMs in Vivado
- Built-in FSM encoding strategies to for optimization

Possible Encoding Strategies :

- Auto
- One-Hot 小
- Gray State 只变一位
- Johnson State 环形移位编码器
- Sequential State 大 顺序

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug901-vivado-synthesis.pdf

> What is FSM inference in Vivado?

✔ 答：

> Vivado can automatically recognize FSM structures from RTL and optimize them using built-in encoding strategies.

> Which FSM encoding is most suitable for FPGA?

✔ 答：

> One-hot encoding, due to fast decoding and abundant flip-flops.

> Does user-defined state encoding force Vivado to use that encoding?

✔ 答：

> No. Vivado may re-encode states internally unless constrained.

八、一句话终极总结（直接背）

> **Vivado 会自动识别 FSM，并根据设计目标选择合适的状态编码；**
> **在 FPGA 中，One-Hot 是最常见的优化选择。**