Lecture 4
Digital System Design with Finite State Machines (FSM)

Prepared by Dr. Shahriyar Masud Rizvi

FSM Definitions and Applications:

Finite State Machines (FSM) or Finite State Automata (FSA) are a type of Automata with a finite number of states. Automaton (plural: automata) is a self-regulating machine that goes through a predetermined sequence of operations without the need for external assistance (that is, it is automatic).

Therefore, FSM is a computational model that goes through a predetermined sequence of operations in a finite number of states.

FSM can be implemented with either hardware or software and is used to design computer programs and digital logic circuits.

In Mathematics and Computer Science, the term Finite State Automata (FSA) is used, while in Electronic and Computer Engineering, the term Finite State Machines (FSM) is used.

FSM is used for applications such as:

- Pattern recognition in text and other data
- Modeling digital circuits
- Designing and analyzing computer algorithms and programs
- Building compilers and interpreters
- Developing language models and machine learning algorithms

The purpose of designing and using FSM is to automate various computational tasks.

FSM is mathematically described by 5 symbols:

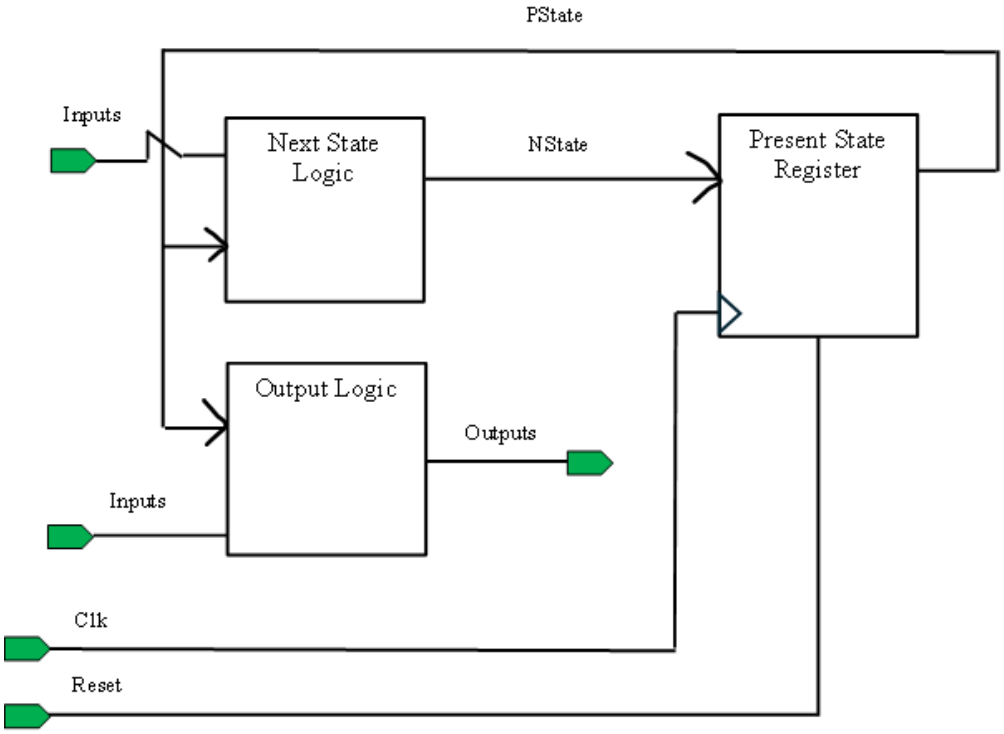
- A finite set of states
- Finite set of inputs
- List of transitions between states
- Initial (starting) state
- Final (accepting) states

In digital logic, FSMs are used to design control circuits (also called control units or controllers).

FSMs can represent any sequential logic circuits.

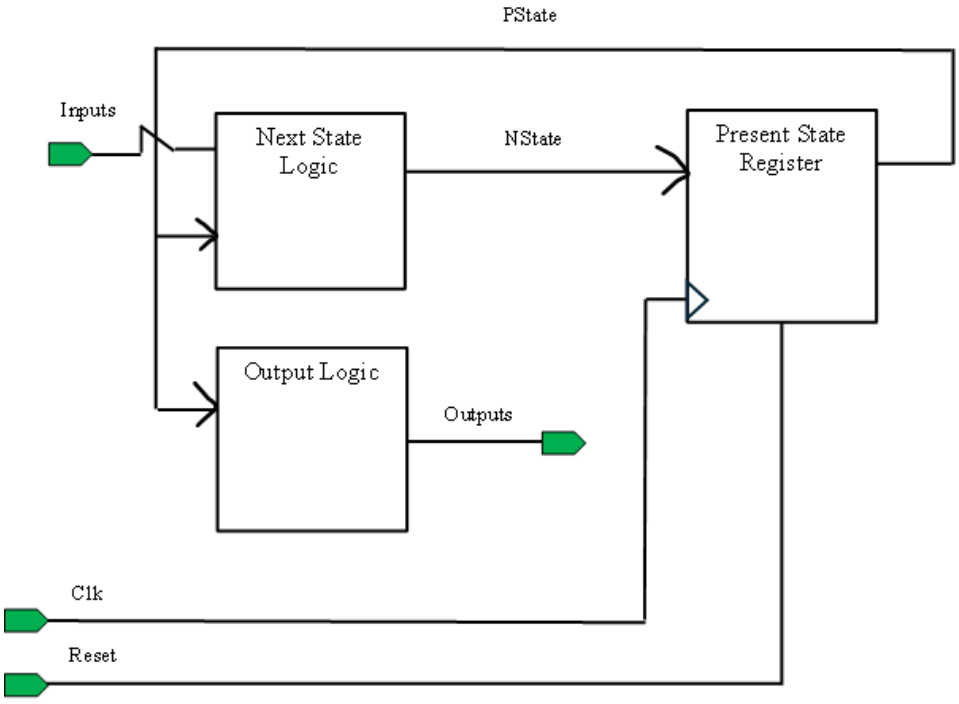
FSM Architecture

FSM is composed of a Present State Register, a next state logic and an output logic, as shown in the following figure. The former is sequential logic and the latter two are combinational logic.

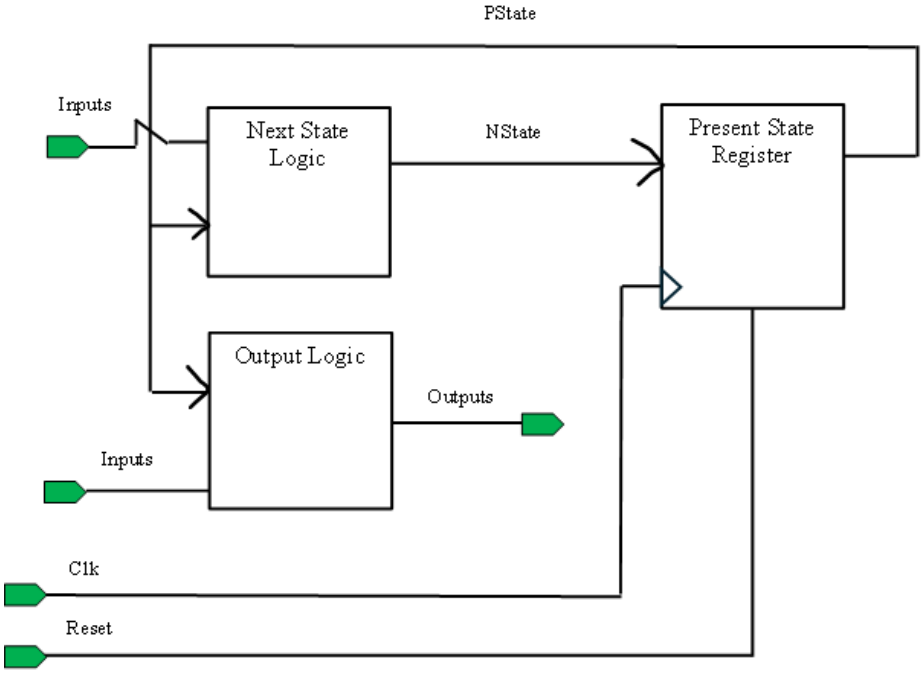


Moore v s Mealy FSMs

FSM can have 2 types of outputs—Moore and Mealy. Moore outputs depend only on present state. Mealy output depends on both present state and inputs. FSMs that produce only Moore outputs are called Moore FSMs. The following figure shows the architecture of Moore FSMs.

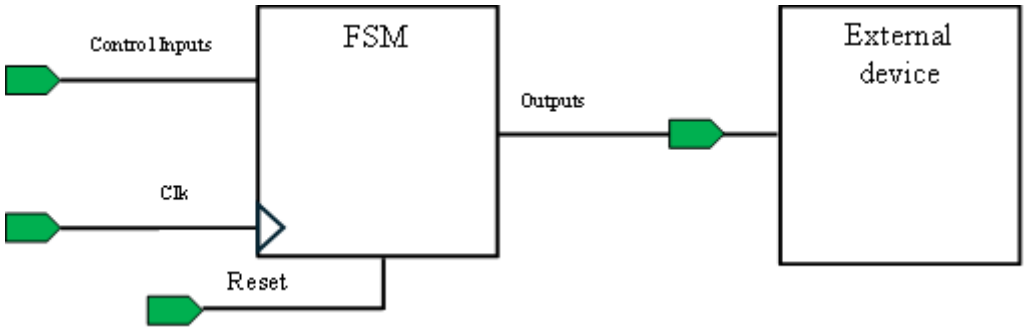


FSMs that produce 1 or more Mealy outputs are called Mealy FSMs. The following figure shows the architecture of Mealy FSMs.



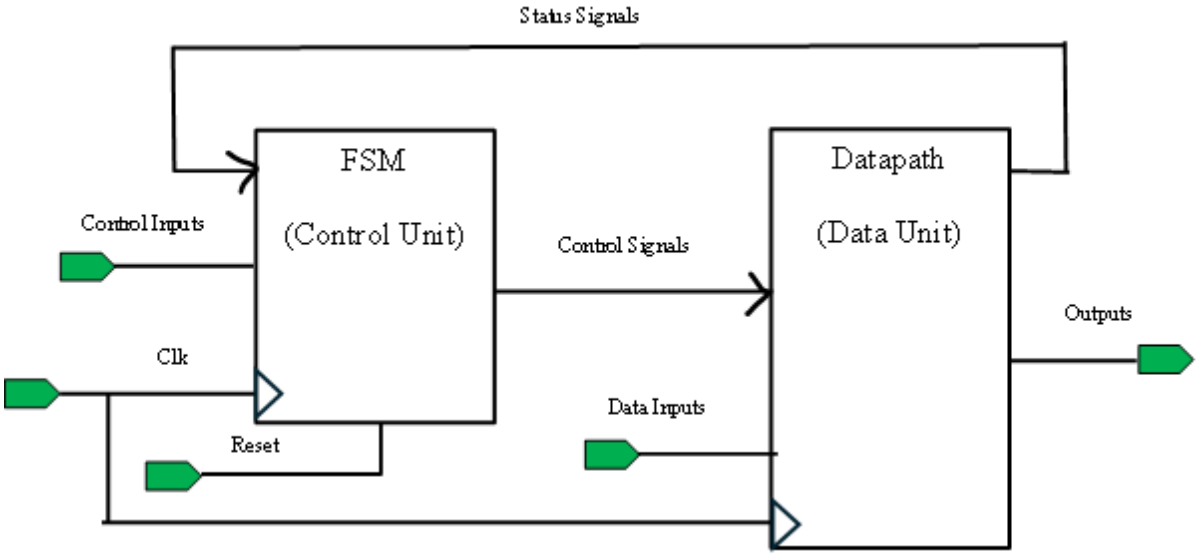
Control-dominated FSMs vs Data-dominated FSMs

In control-dominated designs, such as signal generators (e.g., traffic light controllers) and stepper motor controllers, FSMs control operation of an external device such as LEDs or a stepper motor.



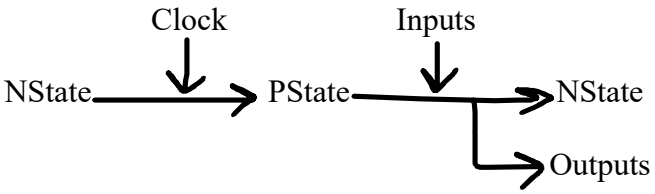
In data-dominated designs, such as a microprocessor, FSMs control logical and arithmetic operations for the computations that a computing device needs to execute. In other words, it controls the operations of computing elements such as Arithmetic and Logic Units (ALU), storage devices for data such as registers and data steering circuits such as multiplexers and demultiplexers. FSM controls these operations through control signals (such as ALU_Control signals, select signals, load/enable signals for ALU, multiplexer and registers, respectively). To ensure proper control of these devices, sometimes the output states of these devices are read back by the FSM as status signals. This is illustrated in the following figure.

Note that the computations are done inside a datapath or data unit. The FSM serves as the control unit that guides the computations so that they are executed correctly and in the right sequence.



Basic FSM Operation

FSM computes the next state signals (NState in the figures) and outputs based on present state signals (PState in the figures) and inputs. When an active clock-edge arrives, the next state values are transferred to the present state signals. When the present state signals are updated, the next state signals and outputs are updated based on the new present state values and the inputs.



State Encoding

FSM states (present states) can be represented or encoded in multiple ways. The obvious one is binary state encoding where states follow a sequential encoding (0, 1, 2, 3…) like a counter. Here, the size of the present state register = ceil($\log_2$(N)), N = no. of states.

In gray state encoding, neighboring states differ by only 1-bit. The size of the present state register = ceil($\log_2$(N)), N = no. of states.

In johnson state encoding, neighboring states also differ by only 1-bit. A chain of states here has an increasing number of 1s or an increasing number of 0s. The size of the present state register = N/2, N = no. of states. So, half the possible states stay unused. For instance, if your FSM has 8 states, you need a 4-bit present state register, but a 4-bit

present state register can have 2^4=16 states. So, 8 states are used, the remaining 8 states are unused.

In one-hot state encoding, only 1 bit is high in a particular state. The size of the present state register = N, N = no. of states. Here, states that have more than one 1 are unused. For instance, if your FSM has 8 states, you need an 8-bit present state register, but an 8-bit present state register can have 2^8=128 states. So, while 8 states are used, the remaining 120 states are unused.

Binary	Gray	Johnson	One-Hot
000	000	0001	00000001
001	001	0011	00000010
010	011	0111	00000100
011	010	1111	00001000
100	110	1110	00010000
101	111	1100	00100000
110	101	1000	01000000
111	100	0000	10000000

Comparison of State Encodings

Binary and Gray state encoding require the smallest sized present state register, hence smallest no. of D flip-flops to store present state. For instance, a binary-encoded FSM with 8 states can be realized with a 3-bit present state register (ceil($\log_2 8$) = 3).

One-Hot state encoding requires the largest sized present state register, hence largest no. of D flip-flops to store present state. For instance, a one-hot-encoded FSM with 8 states needs an 8-bit present state register.

Johnson state encoding requires a smaller present state register than a one-hot-encoded FSM but a larger present state register than a binary or gray-encoded FSM.

Gray and Johnson state encoding are more power-efficient than other state encodings. In these state encodings, during a state change only 1-bit goes through a HIGH-to-LOW or LOW-to-HIGH transition. There is never more than one output transition (High-to-Low or Low-to-High). In CMOS logic, most of the power is dissipated during output transitions. That is why these two state encodings are power-efficient.

Gray and Johnson state encodings have less probability of glitches. In these two encodings only 1-bit goes through a HIGH-to-LOW or LOW-to-HIGH transition during a state change. So, intermediate states do not occur during state change. In contrast, FSM encoded with other state encodings must go through transitions in multiple bits and since fall time and rise time may not be perfectly equal, intermediate states arise. That is why gray encoding is preferred for stepper motor control.

Consider that a stepper motor controller is designed with a binary-encoded FSM and the motor moves in clockwise direction when FSM goes through states 00→01→10→11 and counter-clockwise direction when it goes through states 00→11→10→01. When the stepper motor is

going in clockwise direction, the actual state transitions can happen to be 00→01→**00**→10→11, if fall time and rise time are not exactly equal. The 01 to 10 transition might happen in 2 stages, first there will be a 01 to 00 (High-to-Low) transition in bit-0 and then a 00 to 10 (Low-to-High) transition in bit-1. In this case the stepper motor will briefly rotate in the counter-clockwise direction when it goes through 01 to 00 transition. Because of these issues with binary state encoding, gray encoding is preferred for stepper motor control.

One-hot state encoding allows faster generation of outputs, as each state is uniquely related with 1 bit of present state and that bit can be associated with a specific output. In one-hot state encoding, it is assumed that unused states will not occur. However, practically an error-detecting circuit is needed to ensure that when you are computing next state and output, only 1 bit is uniquely high, number of 1's are not higher than 1.

Consider a simple FSM that produces Ready, Red, Green and Yellow in 4 states. Let us examine the output equations in different state encodings. In certain small designs such as this, one-hot state encoding does not require even gates to compute outputs, certain bits of present state can be directly used as outputs. In binary state encoding, even simple designs such as this require at least 1 AND gate to compute the output.

Present State (if binary encoding is chosen)	**Present State (if one-hot encoding is chosen)**	**Outputs**			
		Ready	**Red**	**Green**	**Yellow**
00	0001	1	0	0	0
01	0010	0	1	0	0
10	0100	0	0	1	0
11	1000	0	0	0	1

Output equations for binary state encoding	**Output equations for one-hot state encoding**
Ready = $\overline{PS1}.\overline{PS0}$	Ready = PS0
Red = $\overline{PS1}.PS0$	Red = PS1
Green = $PS1.\overline{PS0}$	Green = PS2
Yellow = = $PS1.PS0$	Yellow = PS3

In summary, one can summarize the following.

State encoding	**No. of flip-flops (size of present state register)**	**Power consumption**	**Output delay**	**Glitch-free outputs**	**Unused states**
Binary	Low	High	High	No	Small
Gray	Low	Low	High	Yes	Small
Johnson	Moderate	Low	Moderate	Yes	Moderate
One-Hot	Large	High	Low	No	Large

Control-Dominated System

Example 1: Prepare the design for an FSM that stays in an idle state where it generates an output called Ready and evaluates the state of an input called Go. If Go is low, it stays in the idle state. If Go is high, it generates three outputs—Red, Green and Yellow—consecutively (one after another) for 3s, 3s and 1s, respectively. The clock frequency is 1 Hz. **Apply** binary state encoding. **Show** FSM architecture, state chart, state table and logic equations for next states and outputs.

Solution:

Clock f = 1 Hz

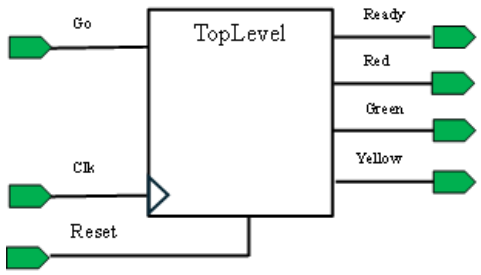
So, Clock T = 1s

Need to output Ready in 1 state, Red in 3 states (Red needs to be high for 3 clock cycles), Green for 3 states (Green needs to be high for 3 clock cycles) and Yellow for 1 state. So, in total there are 8 computational tasks. So, this device needs 8 states.

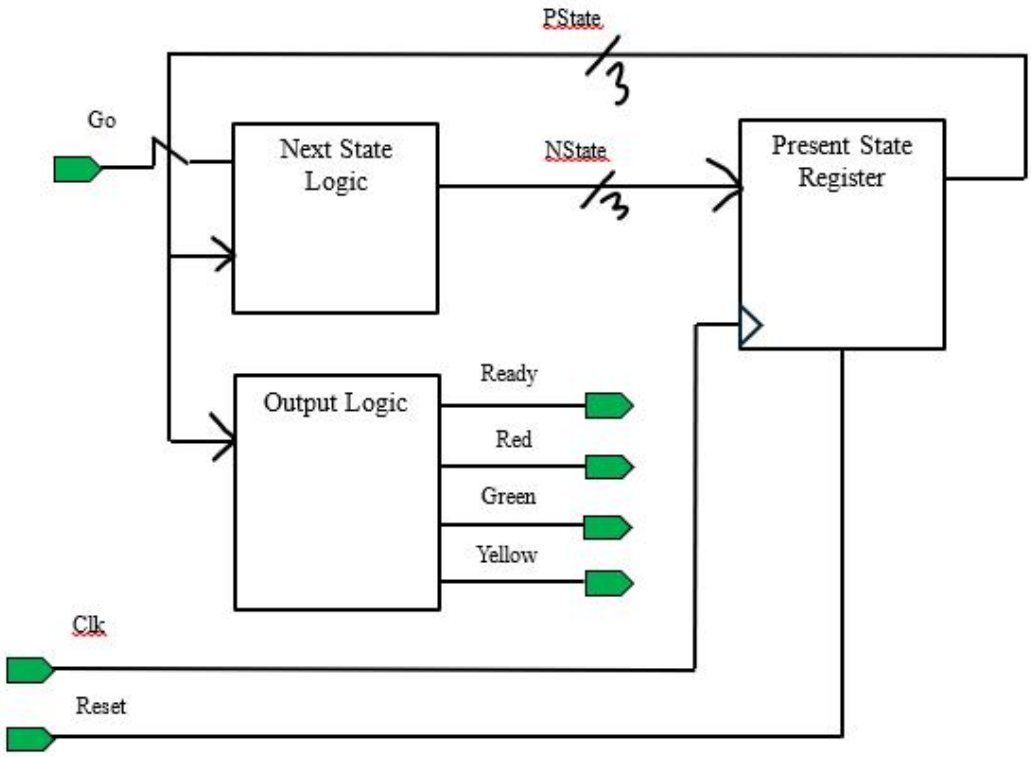
Since binary state encoding is used, the Present State Register must be 3-bit wide (since $\log_2(8) = 3$). In other words, the Present State and Next State signals must be 3-bit wide.

Architecture

Top-level

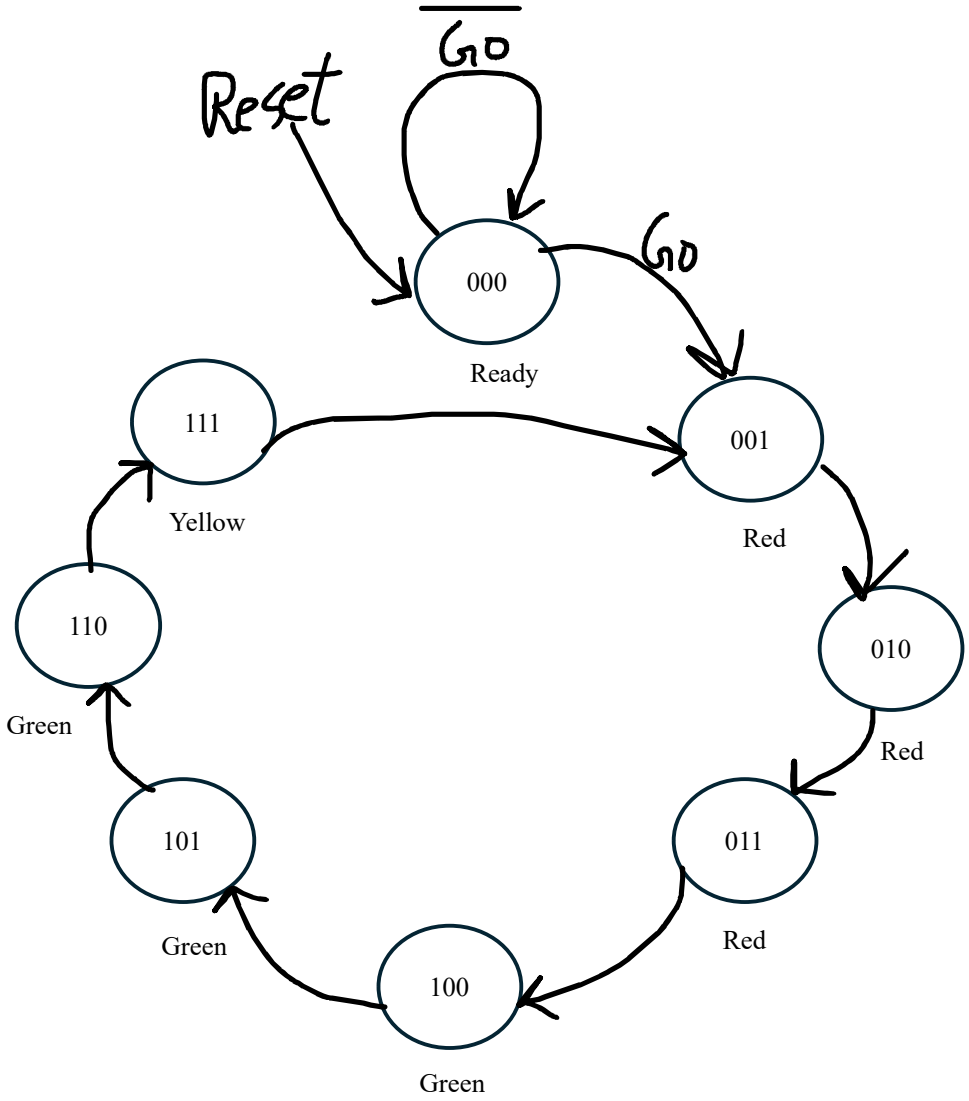


FSM



Algorithm:

State Chart/State Diagram



State Table

Reset	PState	Input	NState	Outputs			
		Go		Ready	Red	Green	Yellow
1	xxxx	x	000	0	0	0	0
0	000	0	000	1	0	0	0
		1	001				
0	001	x	010	0	1	0	0
0	010	x	011	0	1	0	0
0	011	x	100	0	1	0	0
0	100	x	101	0	0	1	0
0	101	x	110	0	0	1	0
0	110	x	111	0	0	1	0
0	111	x	001	0	0	0	1

Logic Equations for NState and Outputs
Using NS and PS as shorthand for NState and PState.

Equation for NState

$NS0 = \overline{Reset}.(\overline{PS2}.\overline{PS1}.\overline{PS0}.Go + \overline{PS2}.PS1.\overline{PS0} + PS2.\overline{PS1}.\overline{PS0} + PS2.PS1.\overline{PS0} + PS2.PS1.PS0)$
$= \overline{Reset}.(\overline{PS2}.\overline{PS1}.\overline{PS0}.Go + \overline{PS0}.(\overline{PS2}.PS1 + PS2.\overline{PS1}) + PS2.PS1.(\overline{PS0} + PS0)$
$= \overline{Reset}.(\overline{PS2}.\overline{PS1}.\overline{PS0}.Go + \overline{PS0}\,(PS2 \oplus PS1) + PS2.PS1)$

$NS1 = \overline{Reset}.(\overline{PS2}.\overline{PS1}.PS0 + \overline{PS2}.PS1.\overline{PS0} + PS2.\overline{PS1}.PS0 + PS2.PS1.\overline{PS0})$
$= \overline{Reset}.(PS0.\overline{PS1}.(\overline{PS2} + PS2) + PS1.\overline{PS0}.(\overline{PS2} + PS2))$
$= \overline{Reset}.(PS0.\overline{PS1} + PS1.\overline{PS0})$
$= \overline{Reset}.(PS0 \oplus PS1)$

$NS2 = \overline{Reset}.(\overline{PS2}.PS1.PS0 + PS2.\overline{PS1}.\overline{PS0} + PS2.\overline{PS1}.PS0 + PS2.PS1.\overline{PS0})$
$= \overline{Reset}.(PS0.(\overline{PS2}.PS1 + PS2.\overline{PS1}) + PS2.\overline{PS0}.(\overline{PS1} + PS1))$
$= \overline{Reset}.(PS0(PS2 \oplus PS1) + PS2.\overline{PS0})$

Equation for Outputs

$Ready = \overline{PS2}.\overline{PS1}.\overline{PS0}$

$Red = \overline{PS2}.\overline{PS1}.PS0 + \overline{PS2}.PS1.\overline{PS0} + \overline{PS2}.PS1.PS0$
$= \overline{PS2}.\overline{PS1}.PS0 + \overline{PS2}.PS1.(\overline{PS0} + PS0)$
$= \overline{PS2}.(\overline{PS1}.PS0 + PS1)$
$= \overline{PS2}.(PS1 + PS0)$

$Green = PS2.\overline{PS1}.\overline{PS0} + PS2.\overline{PS1}.PS0 + PS2.PS1.\overline{PS0}$
$= PS2.\overline{PS1}.(\overline{PS0} + PS0) + PS2.PS1.\overline{PS0}$
$= PS2.(\overline{PS1} + PS1.\overline{PS0})$
$= PS2.(\overline{PS1} + \overline{PS0})$

$Yellow = PS2.PS1.PS0$

Final Equations

NS0 = $\overline{Reset}.(\overline{PS2}.\overline{PS1}.\overline{PS0}.Go + \overline{PS0}\,(PS2 \oplus PS1) + PS2.PS1)$
NS1 = $\overline{Reset}.(PS0 \oplus PS1)$
NS2 = $\overline{Reset}.(PS0(PS2 \oplus PS1) + PS2.\overline{PS0})$

Ready = $\overline{PS2}.\overline{PS1}.\overline{PS0}$
Red = $\overline{PS2}.(PS1 + PS0)$
Green = $PS2.(\overline{PS1} + \overline{PS0})$
Yellow = $PS2.PS1.PS0$

Register Transfer Level (RTL) Description with Verilog HDL

```
module TLC_BINARY
  (input wire CLK, RST, GO,
   output wire READY, RED, GREEN, YELLOW);

//Internal signal definitions
  reg [2:0] P_STATE;
  wire [2:0] N_STATE;

//Next State Logic
  assign N_STATE[0] = ~RST & ( (~P_STATE[2] & ~P_STATE[1] & ~P_STATE[0] & GO )
                             | (~P_STATE[0] & (P_STATE[2] ^ P_STATE[1]) )
                             | (P_STATE[2] & P_STATE[1] ) );

  assign N_STATE[1] =  ~RST & (P_STATE[1] ^ P_STATE[0]);

  assign N_STATE[2] =  ~RST & ( (P_STATE[2] & ~P_STATE[0])
                              | (P_STATE[0] & (P_STATE[2] ^ P_STATE[1]) ) );

//Present State Register
  always @ (posedge CLK)
  begin

    P_STATE[0] <= N_STATE[0];
    P_STATE[1] <= N_STATE[1];
    P_STATE[2] <= N_STATE[2];

  end

//Output Logic

  assign READY      = ~P_STATE[2] & ~P_STATE[1] & ~P_STATE[0];
  assign RED        = ~P_STATE[2] & (P_STATE[1] | P_STATE[0]);
  assign GREEN      = P_STATE[2] & (~P_STATE[1] | ~P_STATE[0]);
  assign YELLOW     = P_STATE[2] & P_STATE[1] & P_STATE[0];

endmodule
```

Final Equations

$NS0 = \overline{Reset}.(\overline{PS2}.\overline{PS1}.\overline{PS0}.Go + \overline{PS0}(PS2 \oplus PS1) + PS2.PS1)$

$NS1 = \overline{Reset}.(PS0 \oplus PS1)$

$NS2 = \overline{Reset}.(PS0(PS2 \oplus PS1) + PS2.\overline{PS0})$

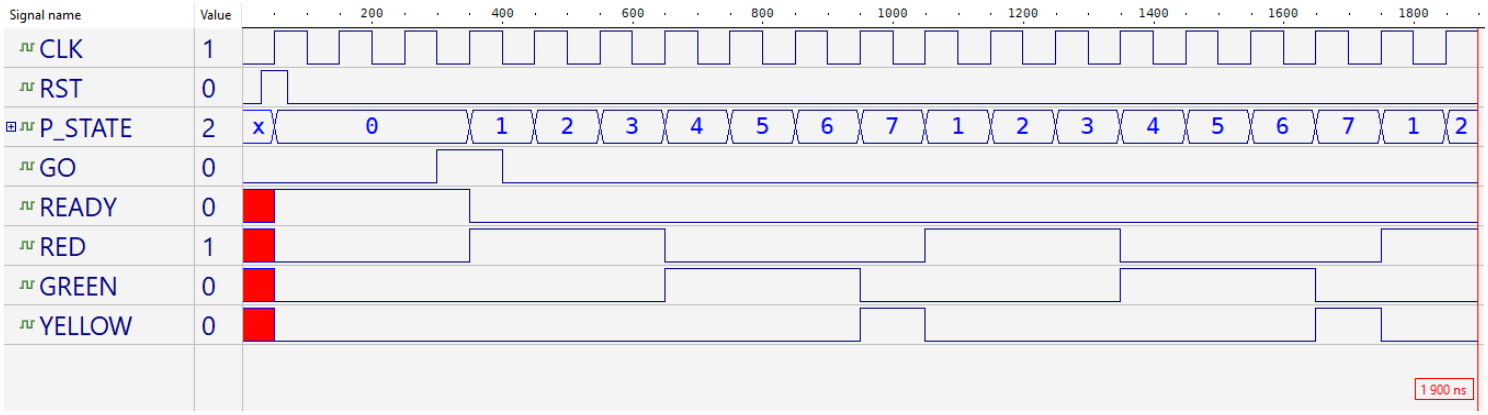
$Ready = \overline{PS2}.\overline{PS1}.\overline{PS0}$

$Red = \overline{PS2}.(PS1 + PS0)$

$Green = PS2.(\overline{PS1} + \overline{PS0})$

$Yellow = PS2.PS1.PS0$

Functional Simulation/RTL Simulation:



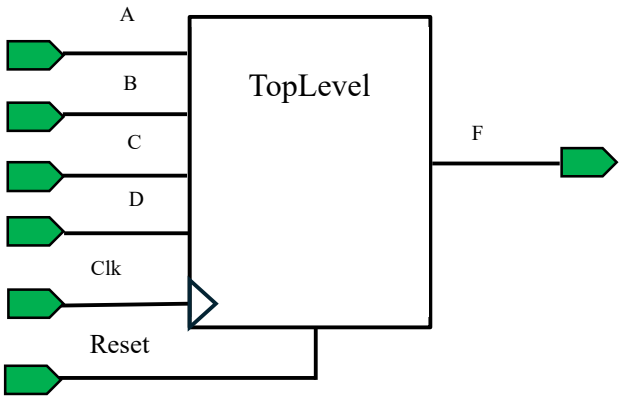
Data-Dominated System

Example 2: Prepare the design for an FSM that controls the following computation: F ←A+B+C+D. You can use 1 adder and 1 register for each calculation step. **Apply** binary state encoding. **Show** Data Flow Graph (DFG), state chart, state table and logic equations for next states and outputs.

Solution:

Architecture

Top-Level:



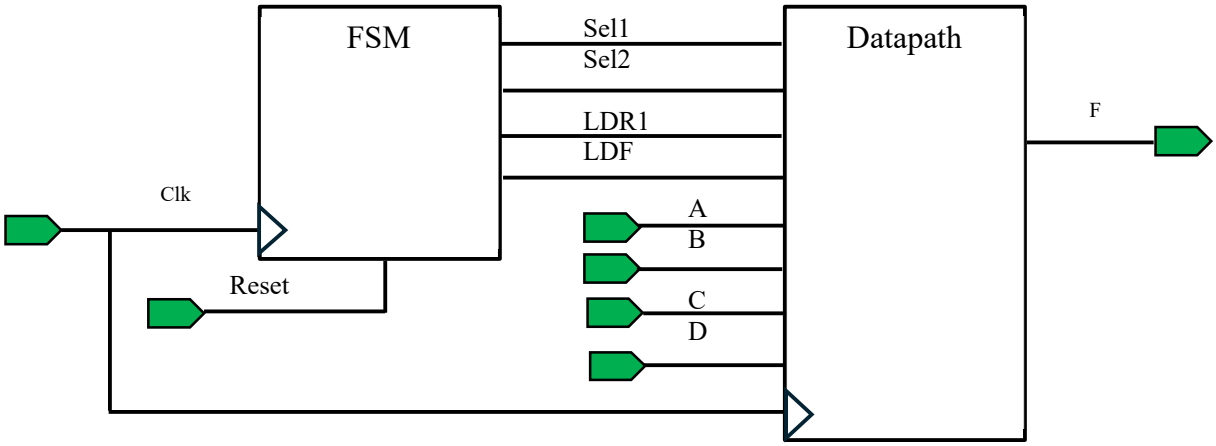
Since only 1 adder can be used, the computation must be done in 3 steps.
Step1 (Clock-Cycle 1): R1←A+B
Step2 (Clock-Cycle 2): R1←R1+C
Step1 (Clock-Cycle 3): F←R1+D

We need 1 adder. We also need 2 multiplexers to provide inputs to this adder. One multiplexer must provide A and R1 to the adder. The other multiplexer must provide B, C and D to the adder.

The multiplexer select signals are labelled Sel1 and Sel2. The load (enable) signals for the registers R1 and F are labelled LDR1 and LDF.

So, the datapath that the FSM is controlling needs to have 1 adder and 2 multiplexers. The FSM would control these datapath elements through the control signals Sel1, Sel2, LDR1 and LDF.

FSM-Datapath Partition

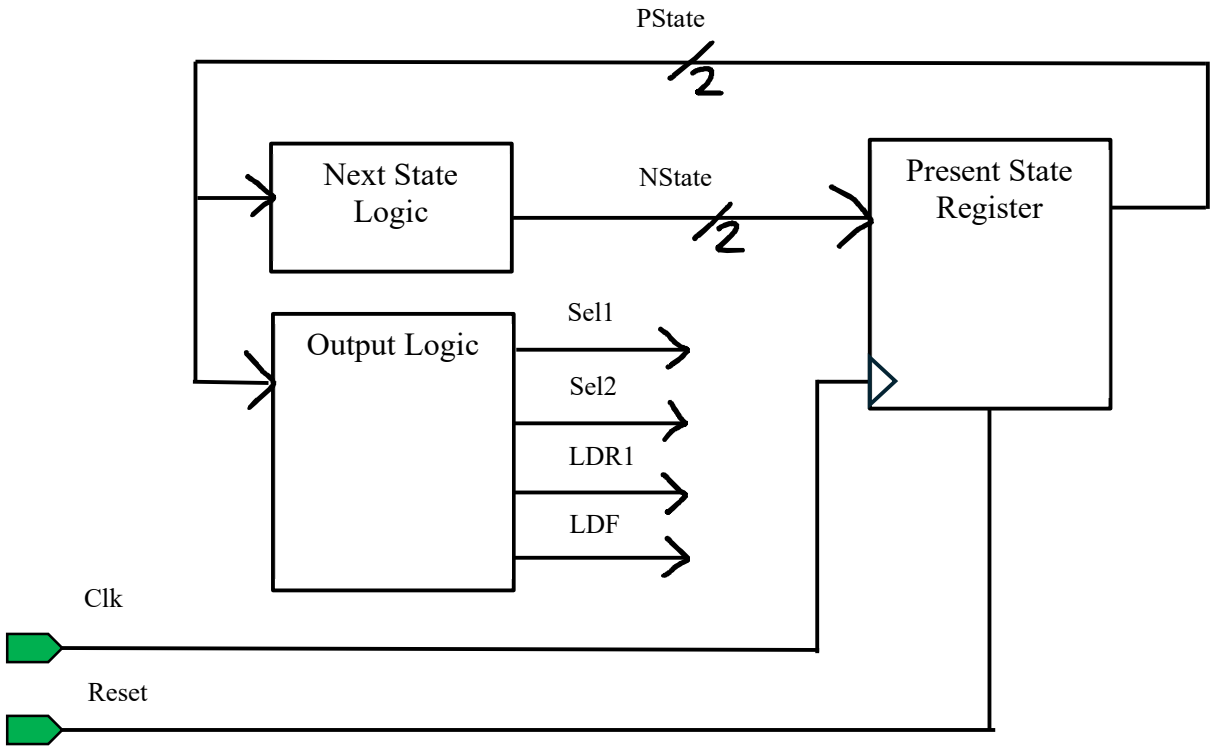


As noted before, only 1 adder can be used, and hence the computation has to be done in 3 steps.
Step1 (Clock-Cycle 1): R1←A+B
Step2 (Clock-Cycle 2): R1←R1+C
Step1 (Clock-Cycle 3): F←R1+D

There are 3 steps. So, the FSM for this device needs 3 states.

Since binary state encoding is used, the Present State Register must be 2-bit wide (since ceil($\log_2(3)$) = ceil(1.58) = 2). In other words, the Present State and Next State signals must be 2-bit wide.

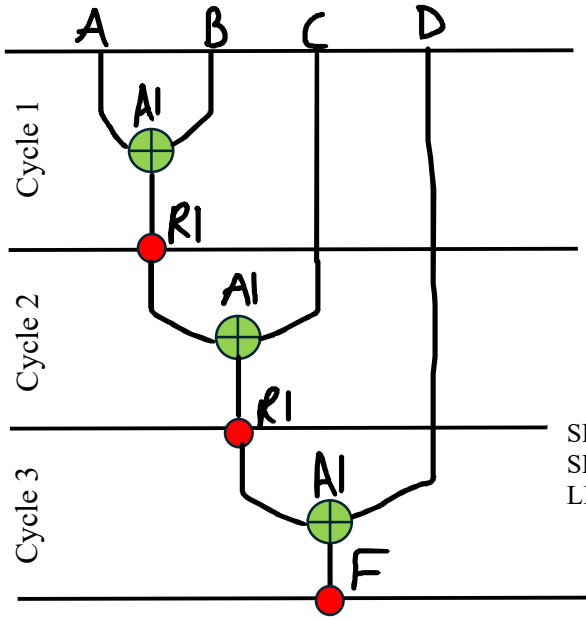
FSM



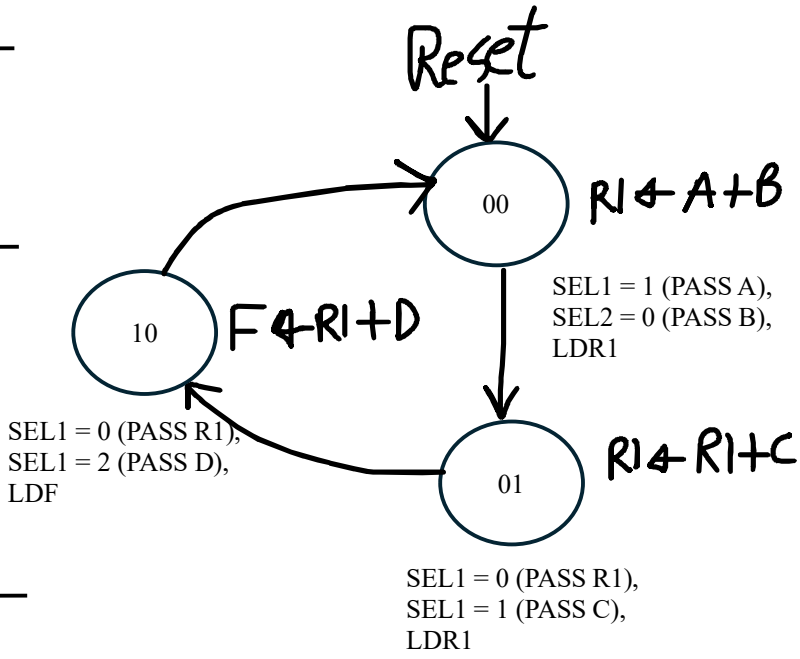
Algorithm & Datapath Architecture:

We can use 1 adder and 1 data register for each calculation step.
Let's name the adder A1 and the register R1.

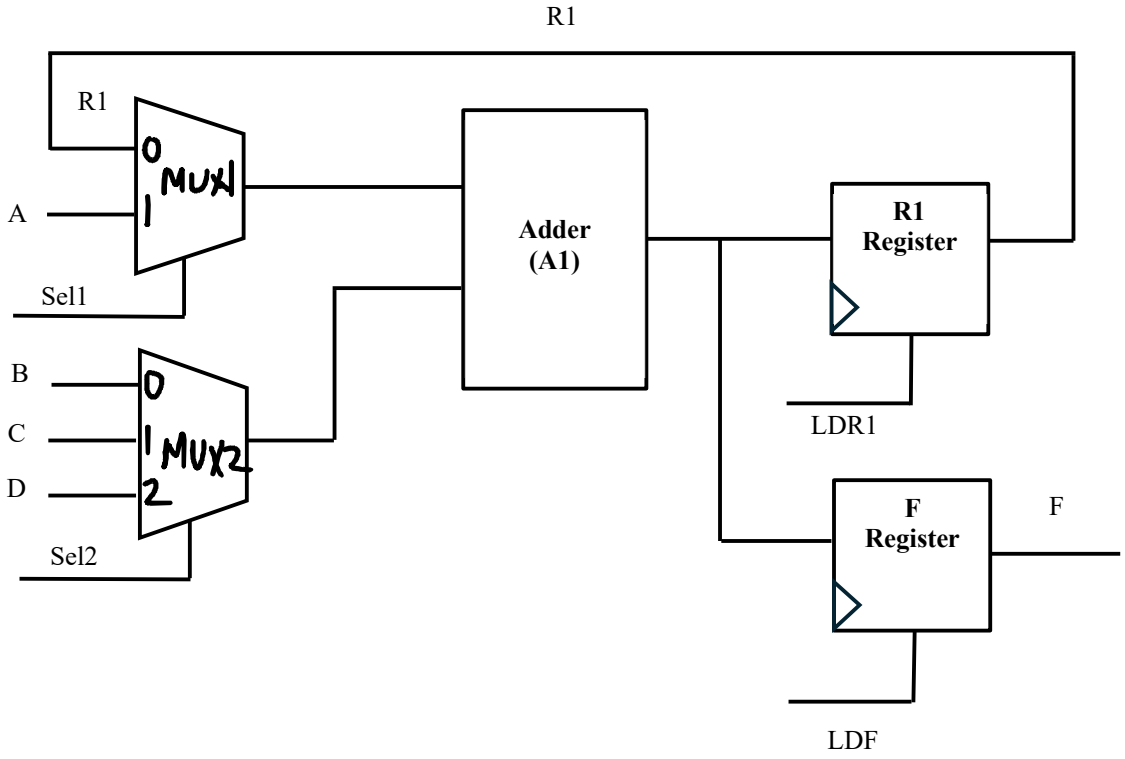
Data Flow Graph (DFG)



State Chart



Datapath Architecture



State Table

Reset	PState	NState	Outputs			
			Sel1	Sel2	LDR1	LDF
1	xx	00	0	00	0	0
0	00	01	1	00	1	0
0	01	10	0	01	1	0
0	10	00	0	10	0	1

Logic Equations

NS0 = $\overline{Reset}.(\overline{PS2}.\overline{PS1})$
NS1 = $\overline{Reset}.(\overline{PS2}.PS1)$

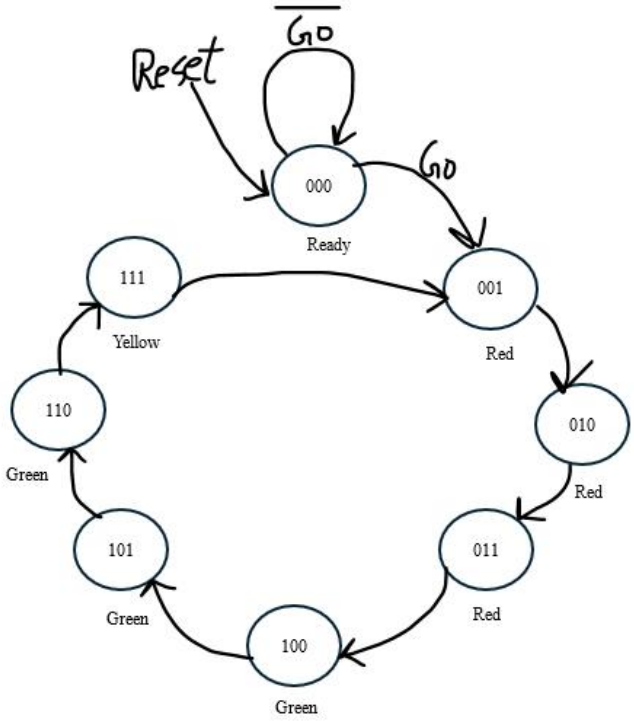
Sel1 = $\overline{Reset}.(\overline{PS2}.\overline{PS1})$

Sel20 = $\overline{Reset}.(\overline{PS2}.PS1)$
Sel21 = $\overline{Reset}.(PS2.\overline{PS1})$

LDR1 = $\overline{Reset}.(\overline{PS2}.\overline{PS1} + \overline{PS2}.PS1)$
LDF = $\overline{Reset}.(PS2.\overline{PS1})$

For Interested Readers

We discussed logic-level (gate-level) Verilog HDL modeling in this course. Modeling at this level of abstraction is not helpful for designing practical digital systems, which are larger and more complex than systems discussed in this course. In practice, behavioral coding style, rather than gate-level coding style is used. The Example 1 design discussed here can be modeled at behavioral level as shown below. The code can be written directly from the state diagram, without any need for state table or logic equations. Note that behavioral code allows use of control structures such as ***if*** and ***case***, that allows easy algorithm-to-RTL translation.




```
module TLC_BEHAVIORAL
(input wire CLK, RST, GO,
  output reg READY, RED, GREEN, YELLOW);

  reg [2:0] P_STATE, N_STATE;

        //NEXT STATE LOGIC AND OUTPUT LOGIC
always @ (P_STATE, GO)
begin
        //NEXT STATE LOGIC
        case(P_STATE)
        3'b000: if (~GO) N_STATE = 3'b000;
                else       N_STATE = 3'b001;
        3'b001: N_STATE = 3'b010;
        3'b010: N_STATE = 3'b011;
        3'b011: N_STATE = 3'b100;
        3'b100: N_STATE = 3'b101;
        3'b101: N_STATE = 3'b110;
        3'b110: N_STATE = 3'b111;
        3'b111: N_STATE = 3'b001;
        default: N_STATE = 3'b000;
        endcase

        //OUTPUT LOGIC
{READY, RED, GREEN, YELLOW} = 'b0;
        case(P_STATE)
        3'b000 :                    READY = 1;
        3'b001, 3'b010, 3'b011 : RED = 1;
        3'b100, 3'b101, 3'b110 : GREEN = 1;
        3'b111 :                    YELLOW = 1;
        default: ;
        endcase
end

//PRESENT STATE REGISTER
always @ (posedge CLK)
begin
        if (RST)
                P_STATE <= 3'b000;
        else
                P_STATE <= N_STATE;
end

endmodule
```

Reference:

1. Michael Ciletti, Advanced Digital Design with the Verilog HDL-2nd Edition, 2010.
2. Doughlas J. Smith, HDL Chip Design-A Practical Guide for Designing, Synthesizing and Simulating ASICs & FPGAs using VHDL or Verilog, 1997.
3. Finite State Machines in Hardware-Theory & Design (with VHDL & SystemVerilog), 2013.