EECS 270 W25
Discussion 8

March 13th, 2025
By: Mick Gordinier

Get LabsLand Open for Today's Discussion!!

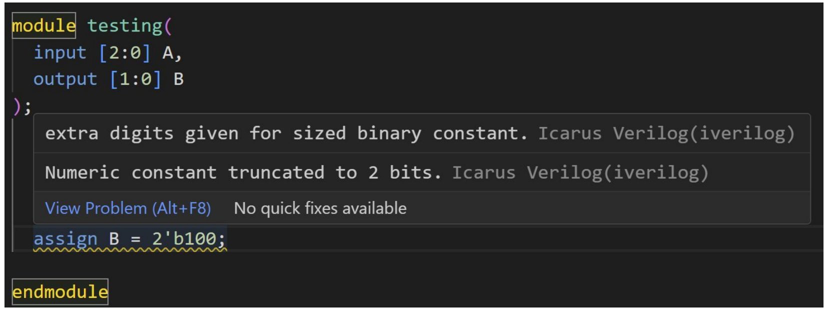
AGENDA

1. Announcements
2. Procedural Statements/Blocks Terminology
3. Sequential Verilog Practice I (Forgiving Lock)
4. Sequential Verilog Practice II (EXTRA-Forgiving Lock)
5. (Bonus) Sequential Verilog Practice III (Self-Selecting MUX)

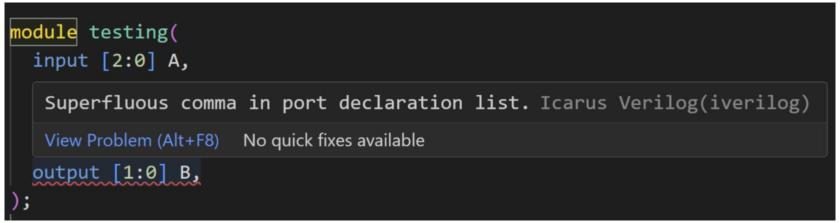
Announcements

Verilog Linter for Windows/Linux

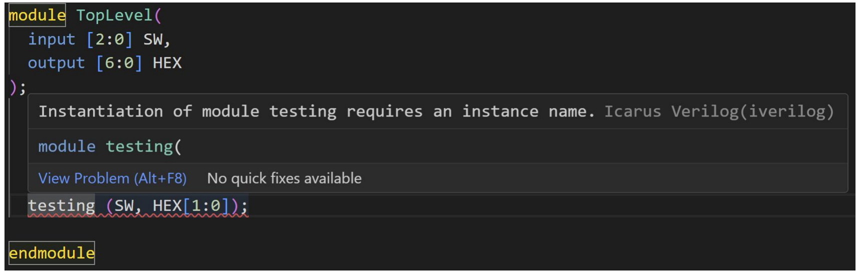
Walkthrough Steps for Installation



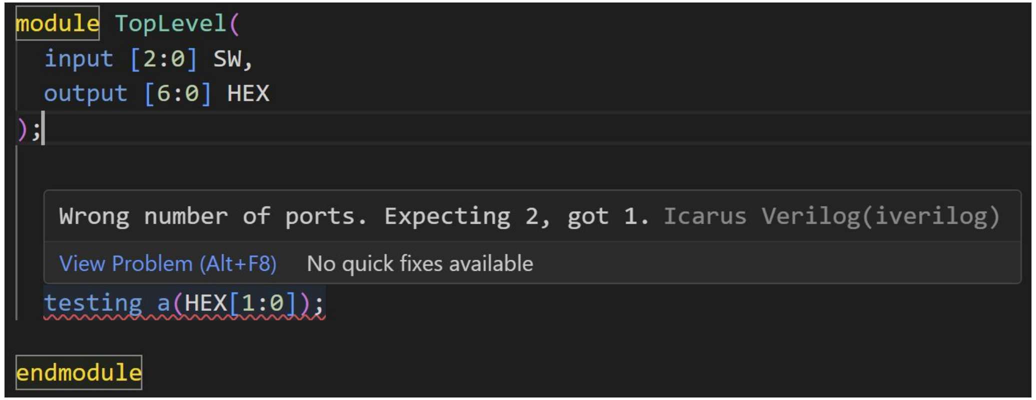
```
module testing(
  input [2:0] A,
  output [1:0] B
);
  extra digits given for sized binary constant. Icarus Verilog(iverilog)
  Numeric constant truncated to 2 bits. Icarus Verilog(iverilog)
  View Problem (Alt+F8)   No quick fixes available
  assign B = 2'b100;

endmodule
```



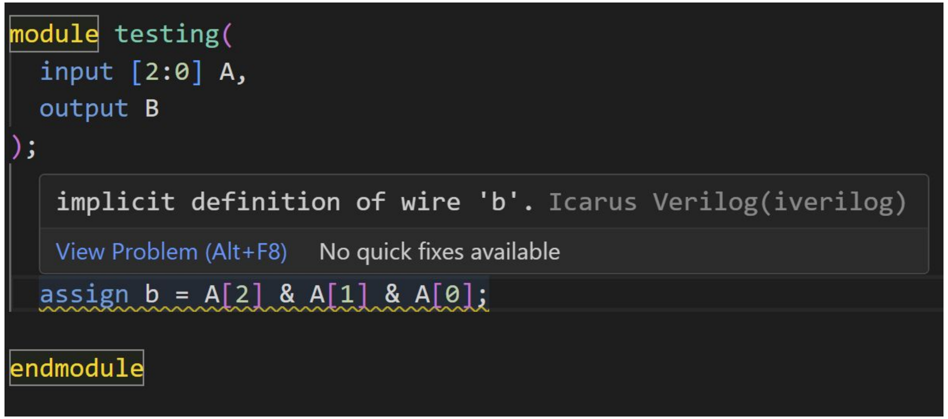
```
module testing(
  input [2:0] A,
  Superfluous comma in port declaration list. Icarus Verilog(iverilog)
  View Problem (Alt+F8)   No quick fixes available
  output [1:0] B,
);
```



```
module TopLevel(
  input [2:0] SW,
  output [6:0] HEX
);
  Instantiation of module testing requires an instance name. Icarus Verilog(iverilog)
  module testing(
  View Problem (Alt+F8)   No quick fixes available
  testing (SW, HEX[1:0]);

endmodule
```



```
module TopLevel(
  input [2:0] SW,
  output [6:0] HEX
);

  Wrong number of ports. Expecting 2, got 1. Icarus Verilog(iverilog)
  View Problem (Alt+F8)   No quick fixes available
  testing a(HEX[1:0]);

endmodule
```



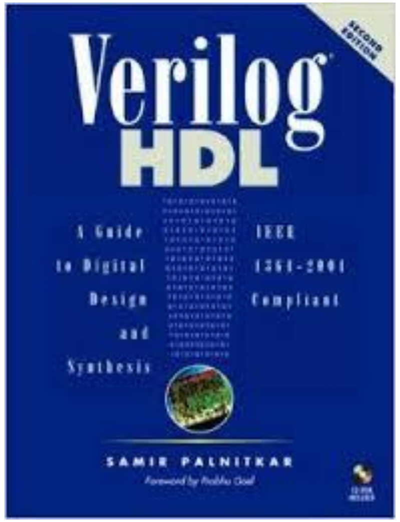
```
module testing(
  input [2:0] A,
  output B
);
  implicit definition of wire 'b'. Icarus Verilog(iverilog)
  View Problem (Alt+F8)   No quick fixes available
  assign b = A[2] & A[1] & A[0];

endmodule
```

Good Reference Verilog Textbook

Verilog HDL: A Guide to Digital Design and Synthesis, Second Edition

- Yes, it is old (2003)

- **I refer to it all the time still**
- Goes over many many great concepts
- Easy to read and understand!



Project 6: Traffic Light Controller

- **WARNING:** This project is extensive (Lots of specification to read)
  - Don’t wait to start it to the last day (Pleaseeeeee)

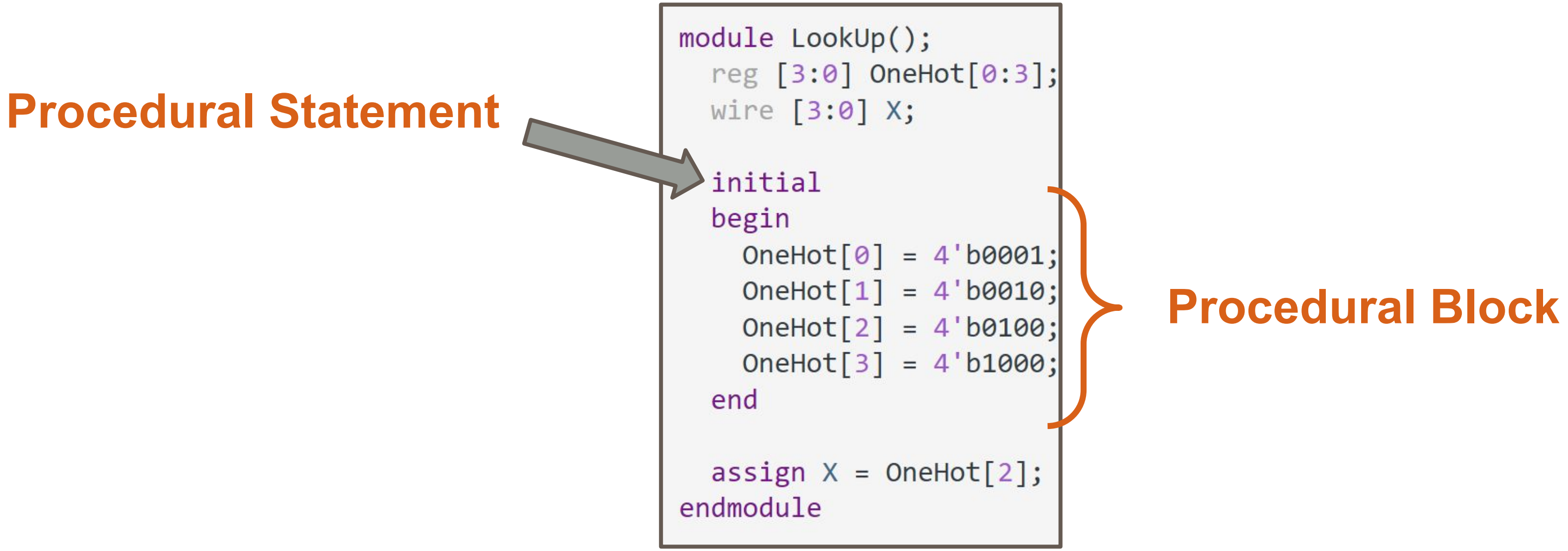
- Please read the specification thoroughly

- Refer back to Modelsim tutorials from previous discussion to get better at making your testbenches

Procedural Statements/Blocks Terminology

Procedural Statements and Blocks

- 2 procedural statements: **always and initial**
  - Each procedural statement represents a separate activity flow
  - All other behavioral modeling statements go inside the procedural blocks



```
module LookUp();
  reg [3:0] OneHot[0:3];
  wire [3:0] X;

  initial
  begin
    OneHot[0] = 4'b0001;
    OneHot[1] = 4'b0010;
    OneHot[2] = 4'b0100;
    OneHot[3] = 4'b1000;
  end

  assign X = OneHot[2];
endmodule
```

“always” Procedural Statements/Blocks

- **Continuously** loops through the statements within the block
- Model activity that is repeated continuously in a digital circuit

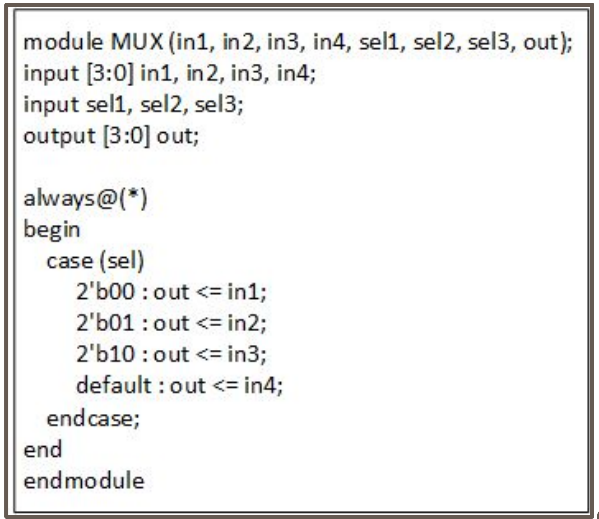
- Allow for complex behavioral statements to be written inside procedural statements
  - Case statements, Conditional Statements, Loops**

Sensitivity List Within “always” Statements

- Sensitivity Lists
  - Way to abstract event-based triggering
  - Only when certain inputs change is when block is triggered

- always @(*)
  - @() - Sensitivity List
  - @(*) - Indicating to watch **all inputs**
  - **If any input changes, trigger block**

- always @(posedge clock)
  - Sequential trigger on posedge of clock



```
module MUX (in1, in2, in3, in4, sel1, sel2, sel3, out);
input [3:0] in1, in2, in3, in4;
input sel1, sel2, sel3;
output [3:0] out;

always@(*)
begin
  case (sel)
     2'b00 : out <= in1;
     2'b01 : out <= in2;
     2'b10 : out <= in3;
     default : out <= in4;
  endcase;
end
endmodule
```

Synthesizability in Verilog

- Remember: Verilog describes the hardware BEHAVIOR
- Your behavior needs to be able to be translated into
  - Hardware Gates, Registers, and RAMs
  - Translation performed by **Synthesizer Tool**

- There exist Verilog code that is **“non-synthesizable”**
  - Not able to be translated into hardware
  - Delay statements, certain data types, $random, …

- Non-synthesizable Verilog exist for **more powerful testbenches**

NANDLAND Src

Notes About Initial Statements (Synthesizability)

- Typically, **initial statement are considered non-synthesizable**
  - Usually ignored during synthesis process

- BUT, Some FPGA synthesizers are exceptions to this (Ours included)

- **“The Quartus® Prime software infers power-up conditions from the Verilog HDL initial constructs”**
  - Initial statements are valid synthesizable Verilog for our FPGA!

Intel Quartus Documentation

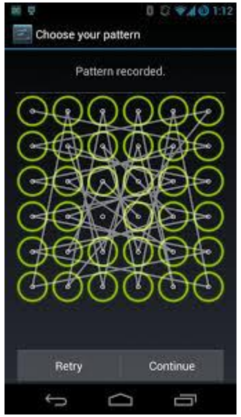
Sequential Verilog Practice I (Forgiving Lock)

Forgiving Lock Motivation

- You are an innovative lockmaker with a passion for craftsmanship, determined to create the next groundbreaking electronic lock!

- **ISSUE:** Sometimes people type in their password wrong!!

Me when I type my password wrong →



- **MY SOLUTION:** Loosen the password "constraints"
  - **Make it easier for user to give correct password!!!**

Me when I type my password RIGHT!!! →

Forgiving Lock Design Problem

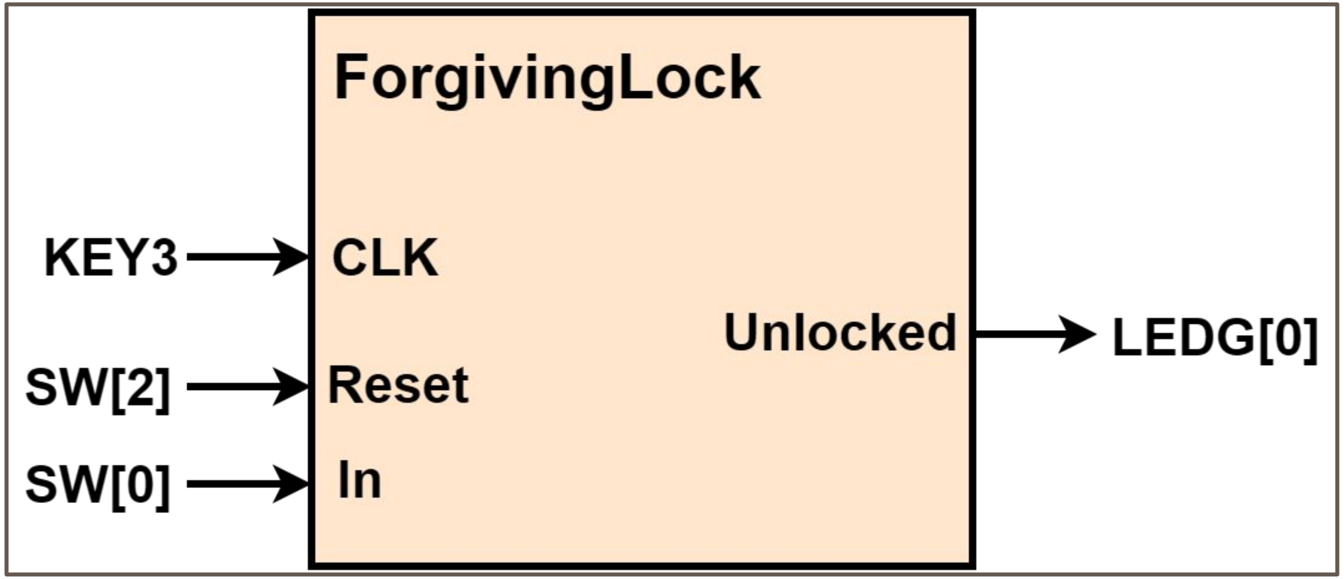
- Allowing users to provide a 4-bit continuous input stream password
  - 0000 –(**1**)→ 000**1** –(**0**)→ 001**0** –(**1**)→ 010**1** –(**1**)→ 101**1** –(**0**)→ 011**0**
  - User must push button to enter new bit into the stream

- User will also have ability to flush password with a priority reset
  - XXXX –(**Reset**)→ **0000**
  - Priority Reset - If reset detected, ignore incoming additional user input

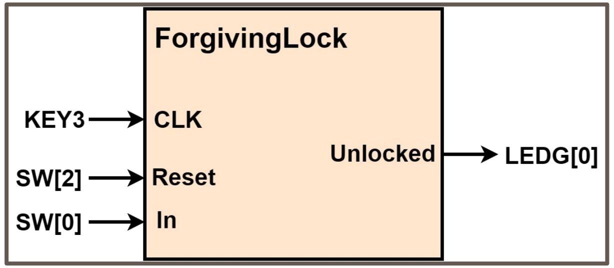
- **Example Loosened Password Constraint**
  - **Allow user to unlock if the current code has alternating numbers in at least 3 digits**
  - **Ex. 010x, 101x, x010, …**

Forgiving Lock Design Diagram

- Will be fully synchronous with a negatively edge triggered active-low button
- Allows 2 input switches: One for Reset, another for Input
- Output: If code is unlocked, light up LEDG[0]



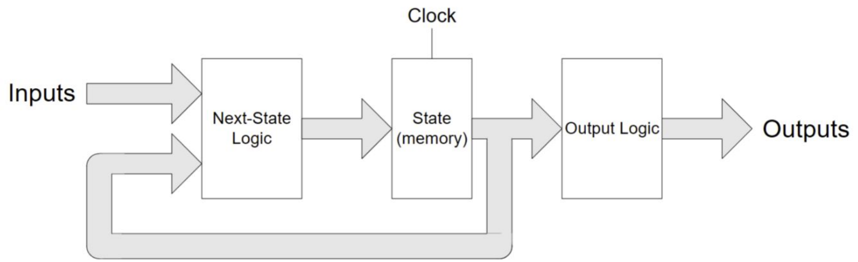
Some Questions



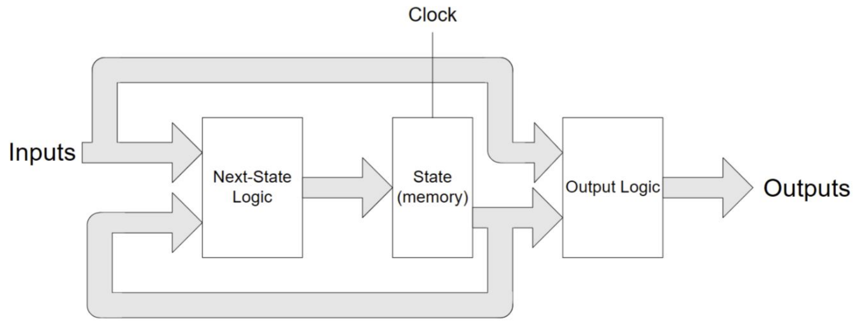
1. **Is this a Moore/Mealy Machine?**
2. **How many internal states exist in this machine?**
3. **For synchronization, should we be using a latch or a FF? Why?**
4. **What does it mean for our machine to be synchronized?**
   a. **If our reset was 'asynchronous', what changes?**
   b. **Why might we choose one over the other?**

Review: Sequential Circuit Components

- **Next state logic** *(combinational)*: next state = f(current state, inputs)
- **Memory** *(sequential)*: stores state in terms of state variables
- **Output logic** *(combinational)*:
  - **Moore Output**: output = g(current state)



  - **Mealy Output**: output= g(current state, inputs)



Construct Transition/Output Table

Code	Input (In)		Unlocked
	0	1	

Code$^+$
(Next_Code)

Code	Input (In)		Unlocked
	0	1	

Code$^+$
(Next_Code)

Construct Transition/Output Table

Code	Input (In)		Unlocked
	0	1	
0000			
0001			
0010			
0011			
0100			
0101			
0110			
0111			

Code$^+$
(Next_Code)

Code	Input (In)		Unlocked
	0	1	
1000			
1001			
1010			
1011			
1100			
1101			
1110			
1111			

Code$^+$
(Next_Code)

Construct Transition/Output Table

Code	Input (In)		Unlocked
	0	1	
0000	0000	0001	
0001			
0010			
0011			
0100			
0101			
0110			
0111			

Code$^+$
(Next_Code)

Code	Input (In)		Unlocked
	0	1	
1000			
1001			
1010			
1011			
1100			
1101			
1110			
1111			

Code$^+$
(Next_Code)

Construct Transition/Output Table

Code	Input (In)		Unlocked
	0	1	
0000	0000	0001	
0001	0010	0011	
0010			
0011			
0100			
0101			
0110			
0111			

Code$^+$
(Next_Code)

Code	Input (In)		Unlocked
	0	1	
1000			
1001			
1010			
1011			
1100			
1101			
1110			
1111			

Code$^+$
(Next_Code)

Construct Transition/Output Table

Code	Input (In)		Unlocked
	0	1	
0000	0000	0001	
0001	0010	0011	
0010	0100	0101	
0011			
0100			
0101			
0110			
0111			

Code$^+$
(Next_Code)

Code	Input (In)		Unlocked
	0	1	
1000			
1001			
1010			
1011			
1100			
1101			
1110			
1111			

Code$^+$
(Next_Code)

Construct Transition/Output Table

Code	Input (In)		Unlocked
	0	1	
0000	0000	0001	
0001	0010	0011	
0010	0100	0101	
0011	0110	0111	
0100			
0101			
0110			
0111			

Code$^+$
(Next_Code)

Code	Input (In)		Unlocked
	0	1	
1000			
1001			
1010			
1011			
1100			
1101			
1110			
1111			

Code$^+$
(Next_Code)

Construct Transition/Output Table

Code	Input (In)		Unlocked
	0	1	
0000	0000	0001	
0001	0010	0011	
0010	0100	0101	
0011	0110	0111	
0100	1000	1001	
0101	1010	1011	
0110	1100	1101	
0111	1110	1111	

Code$^+$
(Next_Code)

Code	Input (In)		Unlocked
	0	1	
1000			
1001			
1010			
1011			
1100			
1101			
1110			
1111			

Code$^+$
(Next_Code)

Construct Transition/Output Table

Code	Input (In)		Unlocked
	0	1	
0000	0000	0001	
0001	0010	0011	
0010	0100	0101	
0011	0110	0111	
0100	1000	1001	
0101	1010	1011	
0110	1100	1101	
0111	1110	1111	

Code^+
(Next_Code)

Code	Input (In)		Unlocked
	0	1	
1000	0000	0001	
1001	0010	0011	
1010	0100	0101	
1011	0110	0111	
1100	1000	1001	
1101	1010	1011	
1110	1100	1101	
1111	1110	1111	

Code^+
(Next_Code)

Construct Transition/Output Table

Code	Input (In)		Unlocked
	0	1	
0000	0000	0001	0
0001	0010	0011	0
0010	0100	0101	1
0011	0110	0111	0
0100	1000	1001	1
0101	1010	1011	1
0110	1100	1101	0
0111	1110	1111	0

Code^{+}
(Next_Code)

Code	Input (In)		Unlocked
	0	1	
1000	0000	0001	0
1001	0010	0011	0
1010	0100	0101	1
1011	0110	0111	1
1100	1000	1001	0
1101	1010	1011	1
1110	1100	1101	0
1111	1110	1111	0

Code^{+}
(Next_Code)

Construct Transition/Output Table

Code	Input (In)		Unlocked
	0	1	
0000	0000	0001	0
0001	0010	0011	0
0010	**0100**	**0101**	**1**
0011	0110	0111	0
0100	**1000**	**1001**	**1**
0101	**1010**	**1011**	**1**
0110	1100	1101	0
0111	1110	1111	0

Code$^+$
(Next_Code)

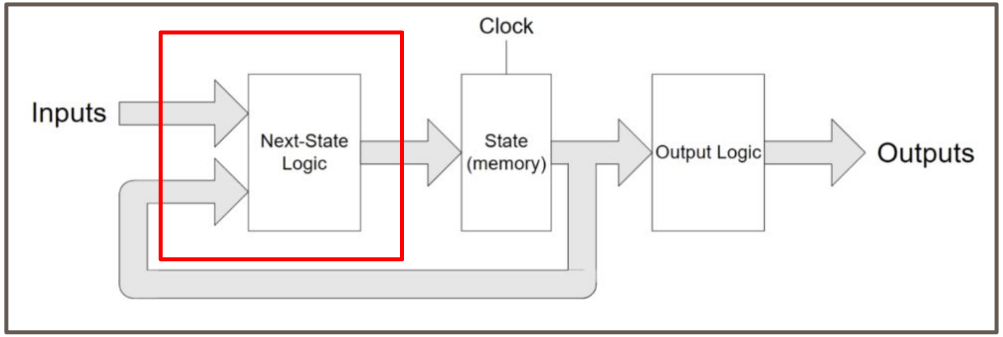
Code	Input (In)		Unlocked
	0	1	
1000	0000	0001	0
1001	0010	0011	0
1010	**0100**	**0101**	**1**
1011	**0110**	**0111**	**1**
1100	1000	1001	0
1101	**1010**	**1011**	**1**
1110	1100	1101	0
1111	1110	1111	0

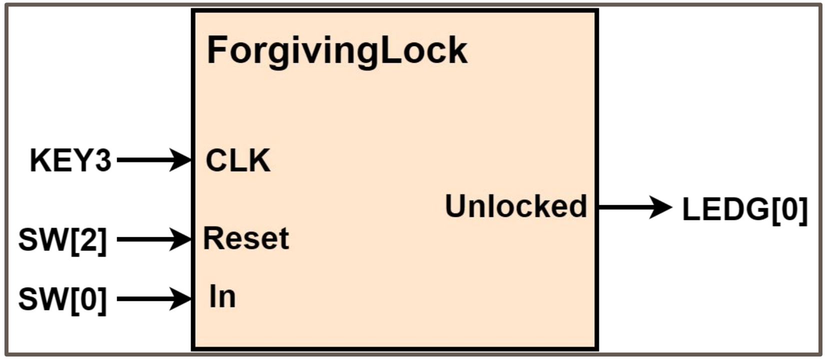
Code$^+$
(Next_Code)

Creating Next State (Transition) Equations

- Need to write equation(s) for Next_Code
    - **Hint: Focus on each bit at a time (How many bits are there?)**
    - **What is Next_Code dependent on?**

- Don't worry about writing code yet





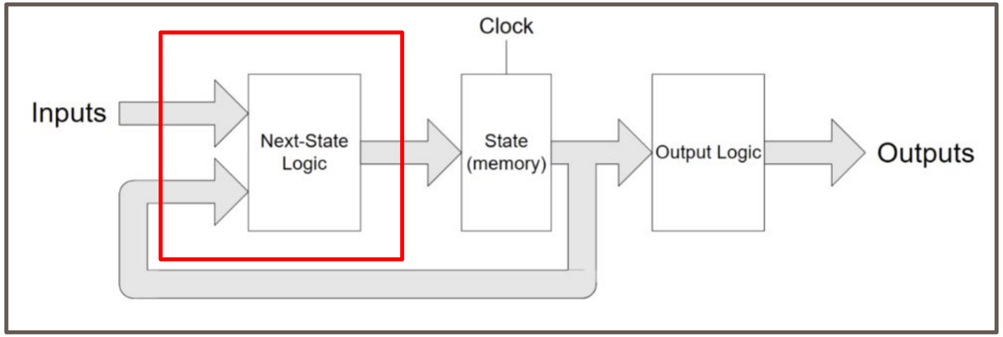
Creating Next State (Transition) Equations

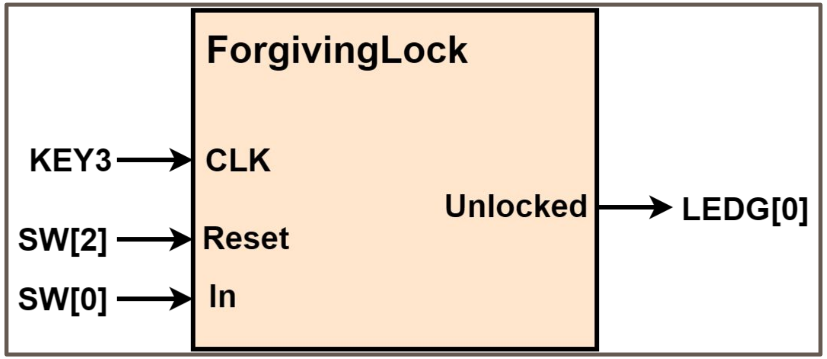
Next_Code[3] = Code[2]

Next_Code[2] = Code[1]

Next_Code[1] = Code[0]

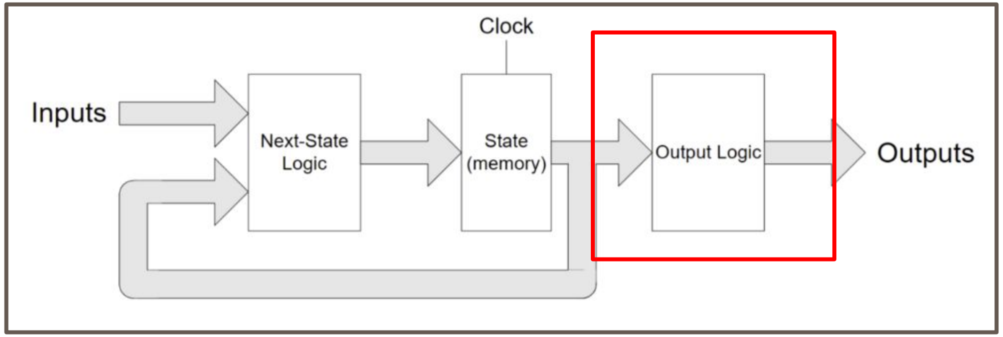
Next_Code[0] = In

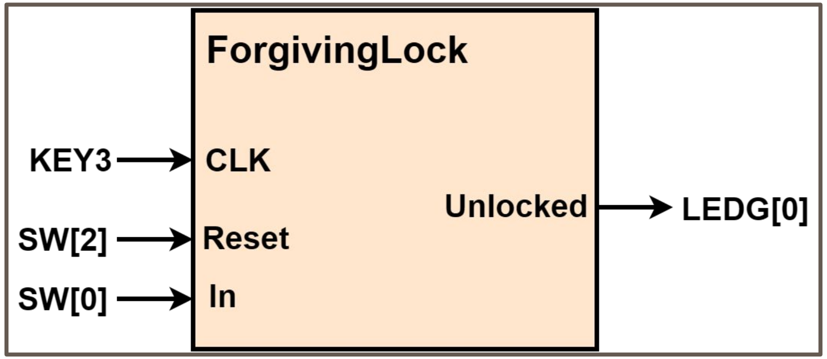




Creating Output Equations

- Need to write an equation for Unlocked
  - **Use a K-Map to get a minimal SOP!!!!**
  - **Remember: We are dealing with a Moore machine**





Creating Output Equations (K-Map Approach)

{Code[3], Code[2]} / {Code[1], Code[0]}				

Creating Output Equations (K-Map Approach)

{Code[3], Code[2]} / {Code[1], Code[0]}	00	01	11	10
00				
01				
11				
10				

Creating Output Equations (K-Map Approach)

{Code[3], Code[2]} / {Code[1], Code[0]}	00	01	11	10
00	0			
01				
11				
10				

Creating Output Equations (K-Map Approach)

{Code[3], Code[2]} / {Code[1], Code[0]}	**00**	**01**	**11**	**10**
00	0	0		
01				
11				
10				

Creating Output Equations (K-Map Approach)

{Code[3], Code[2]} / {Code[1], Code[0]}	00	01	11	10
00	0	0	0	
01				
11				
10				

Creating Output Equations (K-Map Approach)

{Code[3], Code[2]} / {Code[1], Code[0]}	**00**	**01**	**11**	**10**
00	0	0	0	1
01				
11				
10				

Creating Output Equations (K-Map Approach)

{Code[3], Code[2]} / {Code[1], Code[0]}	00	01	11	10
00	0	0	0	1
01	1	1	0	0
11				
10				

Creating Output Equations (K-Map Approach)

{Code[3], Code[2]} / {Code[1], Code[0]}	00	01	11	10
00	0	0	0	1
01	1	1	0	0
11	0	1	0	0
10	0	0	1	1

Creating Output Equations (K-Map Approach)

{Code[3], Code[2]} / {Code[1], Code[0]}	00	01	11	10
00	0	0	0	1
01	1	1	0	0
11	0	1	0	0
10	0	0	1	1

Creating Output Equations (K-Map Approach)

{Code[3], Code[2]} / {Code[1], Code[0]}	00	01	11	10
00	0	0	0	1
01	1	1	0	0
11	0	1	0	0
10	0	0	1	1

1. **Find all Prime Implicants!**

Creating Output Equations (K-Map Approach)

{Code[3], Code[2]} / {Code[1], Code[0]}	00	01	11	10
00	0	0	0	1
01	1	1	0	0
11	0	1	0	0
10	0	0	1	1

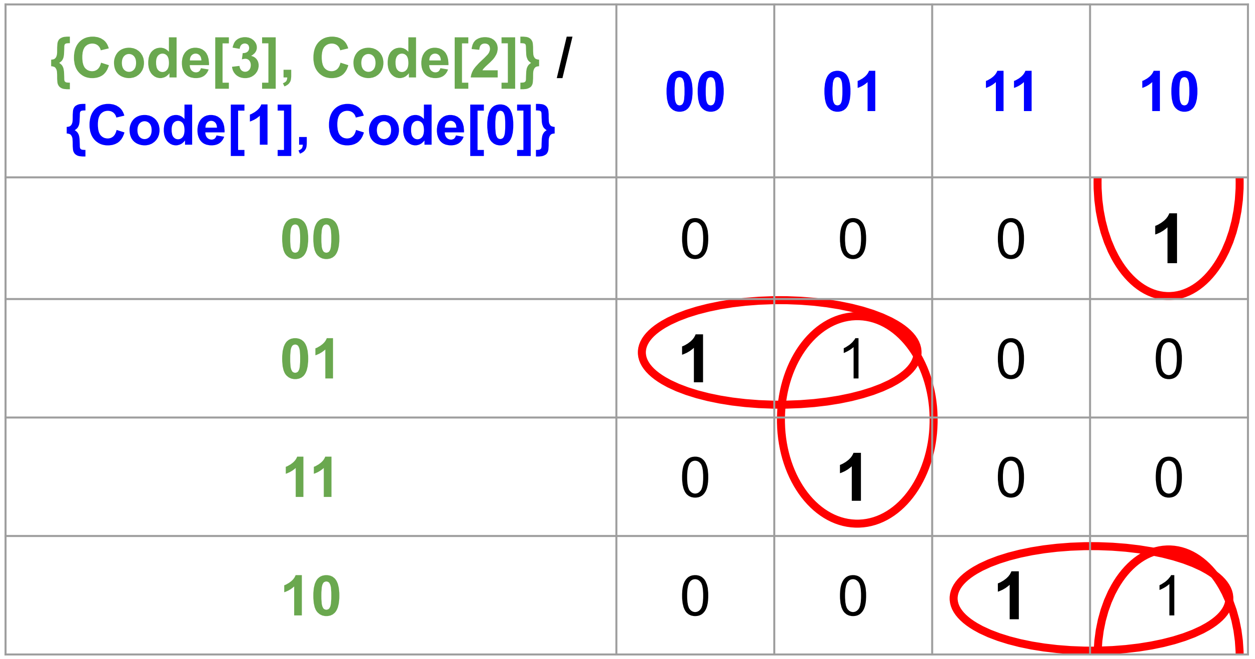
1. **Find all Prime Implicants!**

Creating Output Equations (K-Map Approach)

{Code[3], Code[2]} / {Code[1], Code[0]}	00	01	11	10
00	0	0	0	1
01	1	1	0	0
11	0	1	0	0
10	0	0	1	1

1. **Find all Prime Implicants!**
2. **Indicate which PI are Essential (EPI)**

Creating Output Equations (K-Map Approach)



{Code[3], Code[2]} / {Code[1], Code[0]}	00	01	11	10
00	0	0	0	**1**
01	**1**	1	0	0
11	0	**1**	0	0
10	0	0	**1**	1

1. **Find all Prime Implicants!**
2. **Indicate which PI are Essential (EPI)**

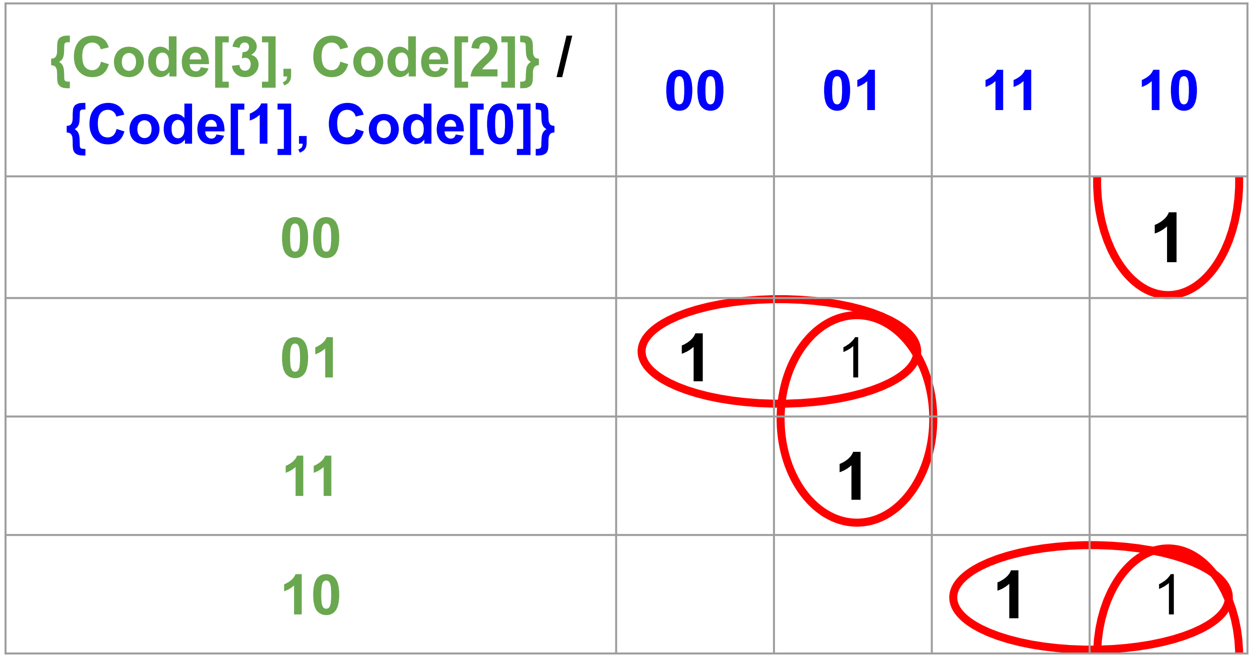
Creating Output Equations (K-Map Approach)

{Code[3], Code[2]} / {Code[1], Code[0]}	00	01	11	10
00	0	0	0	**1**
01	**1**	1	0	0
11	0	**1**	0	0
10	0	0	**1**	1

1. **Find all Prime Implicants!**
2. **Indicate which PI are Essential (EPI)**
3. **Remove all EPIs and remove any dominated PIs**

Unlocked =

Creating Output Equations (K-Map Approach)

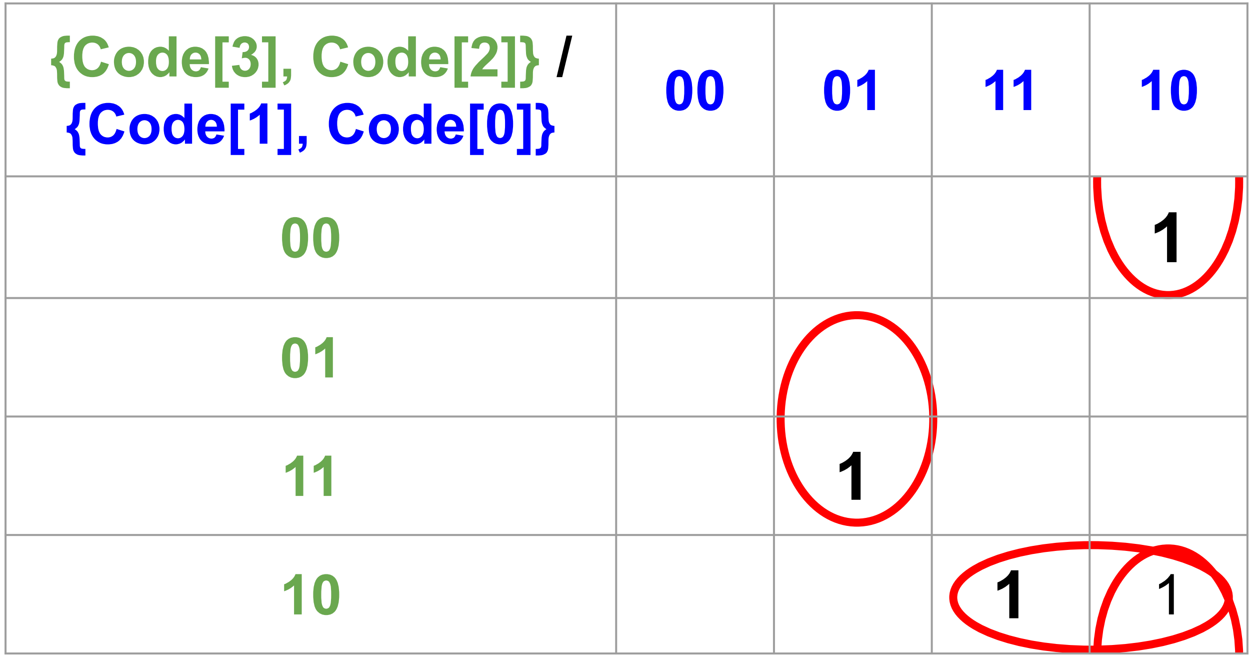


{Code[3], Code[2]} / {Code[1], Code[0]}	00	01	11	10
00				1
01	1	1		
11		1		
10			1	1

Unlocked =

1. **Find all Prime Implicants!**
2. **Indicate which PI are Essential (EPI)**
3. **Remove all EPIs and remove any dominated PIs**

Creating Output Equations (K-Map Approach)



{Code[3], Code[2]} / {Code[1], Code[0]}	00	01	11	10
00				1
01				
11		1		
10			1	1

1. **Find all Prime Implicants!**
2. **Indicate which PI are Essential (EPI)**
3. **Remove all EPIs and remove any dominated PIs**

Unlocked = (~Code[3] & Code[2] & Code[1])

Creating Output Equations (K-Map Approach)

{Code[3], Code[2]} / {Code[1], Code[0]}	00	01	11	10
00				1
01				
11				
10			1	1

1. **Find all Prime Implicants!**
2. **Indicate which PI are Essential (EPI)**
3. **Remove all EPIs and remove any dominated PIs**

Unlocked = (~Code[3] & Code[2] & ~Code[1]) | (Code[2] & ~Code[1] & Code[0])

Creating Output Equations (K-Map Approach)

{Code[3], Code[2]} / {Code[1], Code[0]}	00	01	11	10
00				1
01				
11				
10				

1. **Find all Prime Implicants!**
2. **Indicate which PI are Essential (EPI)**
3. **Remove all EPIs and remove any dominated PIs**

Unlocked = (~Code[3] & Code[2] & ~Code[1]) | (Code[2] & ~Code[1] & Code[0]) | (Code[3] & ~Code[2] & Code[1])

Creating Output Equations (K-Map Approach)

{Code[3], Code[2]} / {Code[1], Code[0]}	00	01	11	10
00				
01				
11				
10				

1. **Find all Prime Implicants!**
2. **Indicate which PI are Essential (EPI)**
3. **Remove all EPIs and remove any dominated PIs**

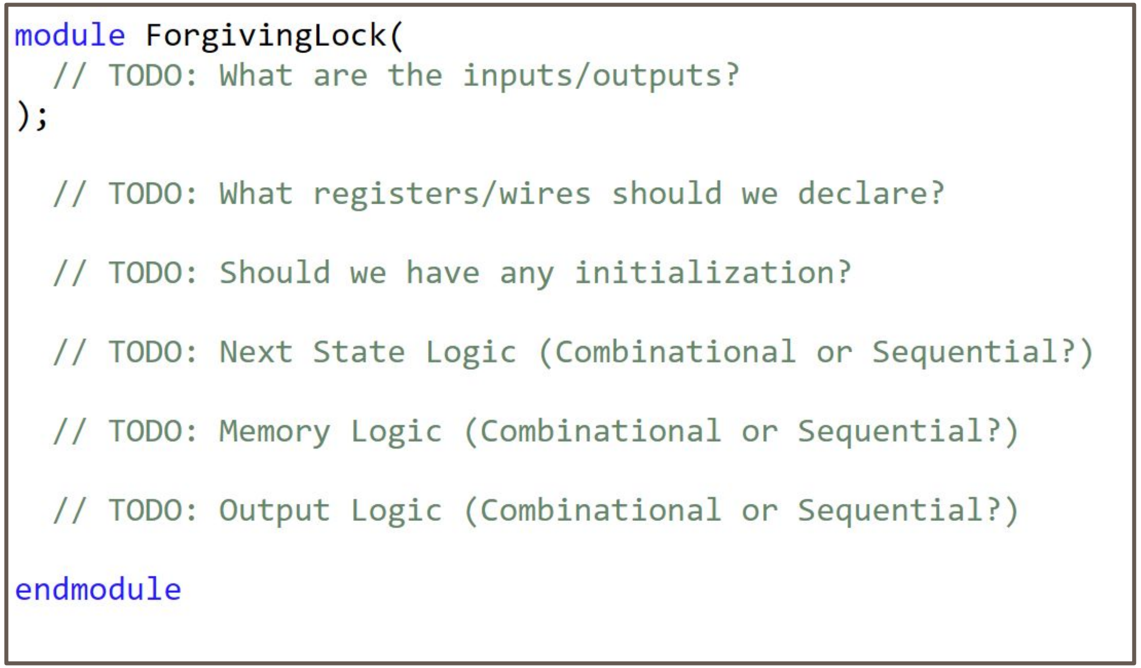
Unlocked = (~Code[3] & Code[2] & ~Code[1]) | (Code[2] & ~Code[1] & Code[0]) | (Code[3] & ~Code[2] & Code[1]) | (~Code[2] & Code[1] & ~Code[0])

Creating Output Equations (K-Map Approach)

```
Unlocked = (~Code[3] & Code[2] & ~Code[1])  |
           (Code[2] & ~Code[1] & Code[0])   |
           (Code[3] & ~Code[2] & Code[1])   |
           (~Code[2] & Code[1] & ~Code[0])
```

1. Find all Prime Implicants!
2. Indicate which PI are Essential (EPI)
3. Remove all EPIs and remove any dominated PIs
4. DONE in 1 round!!

Starter Code Template **(Code to Copy on Right)**

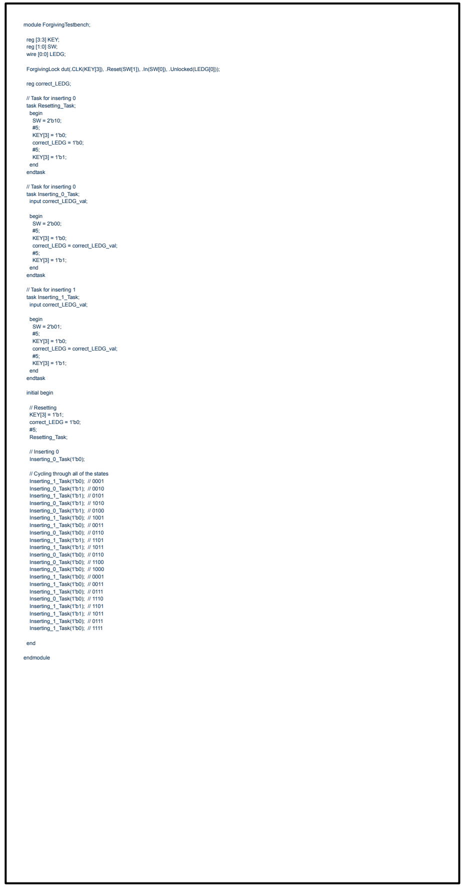


```
module ForgivingLock(
  // TODO: What are the inputs/outputs?
);

  // TODO: What registers/wires should we declare?

  // TODO: Should we have any initialization?

  // TODO: Next State Logic (Combinational or Sequential?)

  // TODO: Memory Logic (Combinational or Sequential?)

  // TODO: Output Logic (Combinational or Sequential?)

endmodule
```

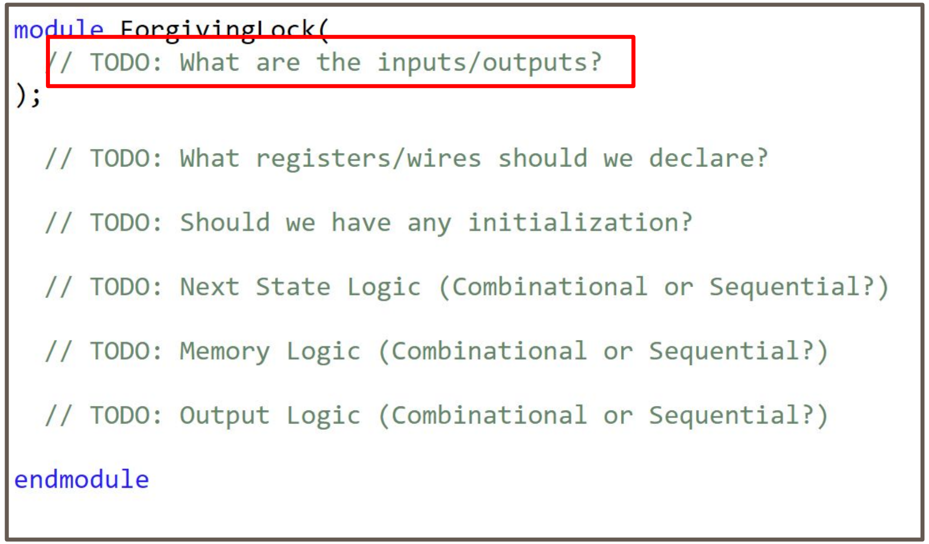
```
module ForgivingLock(
  // TODO: What are the inputs/outputs?
);

  // TODO: What registers/wires should we declare?

  // TODO: Should we have any initialization?

  // TODO: Next State Logic (Combinational or Sequential?)

  // TODO: Memory Logic (Combinational or Sequential?)

  // TODO: Output Logic (Combinational or Sequential?)

endmodule
```

Top-Level Module and Testbench Code

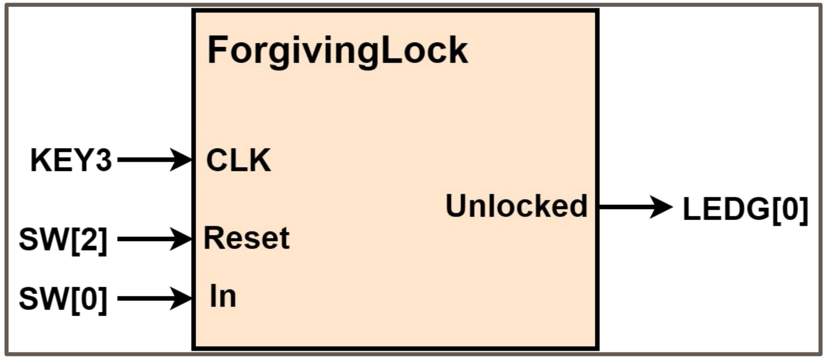
```
module ForgivingTopLevel(
  input [3:3] KEY,
  input [1:0] SW,
  output [0:0] LEDG
);

  ForgivingLock i(.CLK(KEY[3]), .Reset(SW[1]), .In(SW[0]), .Unlocked(LEDG[0]));

endmodule
```



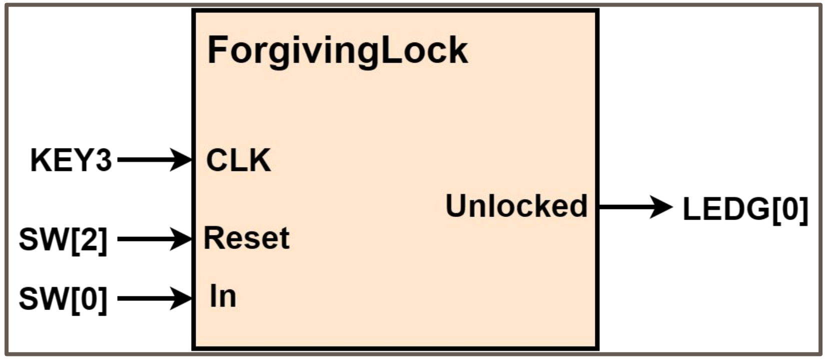
```
module ForgivingTestbench;

  reg [3:3] KEY;
  reg [1:0] SW;
  wire [0:0] LEDG;

  ForgivingLock dut(.CLK(KEY[3]), .Reset(SW[1]), .In(SW[0]), .Unlocked(LEDG[0]));

  reg correct_LEDG;

  // Task for inserting 0
  task Resetting_Task;
    begin
      SW = 2'b10;
      #5;
      KEY[3] = 1'b0;
      correct_LEDG = 1'b0;
      #5;
      KEY[3] = 1'b1;
    end
  endtask

  // Task for inserting 0
  task Inserting_0_Task;
    input correct_LEDG_val;

    begin
      SW = 2'b00;
      #5;
      KEY[3] = 1'b0;
      correct_LEDG = correct_LEDG_val;
      #5;
      KEY[3] = 1'b1;
    end
  endtask

  // Task for inserting 1
  task Inserting_1_Task;
    input correct_LEDG_val;

    begin
      SW = 2'b01;
      #5;
      KEY[3] = 1'b0;
      correct_LEDG = correct_LEDG_val;
      #5;
      KEY[3] = 1'b1;
    end
  endtask

  initial begin

    // Resetting
    KEY[3] = 1'b1;
    correct_LEDG = 1'b0;
    #5;
    Resetting_Task;

    // Inserting 0
    Inserting_0_Task(1'b0);

    // Cycling through all of the states
    Inserting_1_Task(1'b0);  // 0001
    Inserting_0_Task(1'b1);  // 0010
    Inserting_1_Task(1'b1);  // 0101
    Inserting_0_Task(1'b1);  // 1010
    Inserting_0_Task(1'b1);  // 0100
    Inserting_1_Task(1'b0);  // 1001
    Inserting_1_Task(1'b0);  // 0011
    Inserting_0_Task(1'b0);  // 0110
    Inserting_1_Task(1'b1);  // 1101
    Inserting_1_Task(1'b1);  // 1011
    Inserting_0_Task(1'b0);  // 0110
    Inserting_0_Task(1'b0);  // 1100
    Inserting_0_Task(1'b0);  // 1000
    Inserting_1_Task(1'b0);  // 0001
    Inserting_1_Task(1'b0);  // 0011
    Inserting_1_Task(1'b0);  // 0111
    Inserting_0_Task(1'b0);  // 1110
    Inserting_1_Task(1'b1);  // 1101
    Inserting_1_Task(1'b1);  // 1011
    Inserting_1_Task(1'b0);  // 0111
    Inserting_1_Task(1'b0);  // 1111

  end

endmodule
```



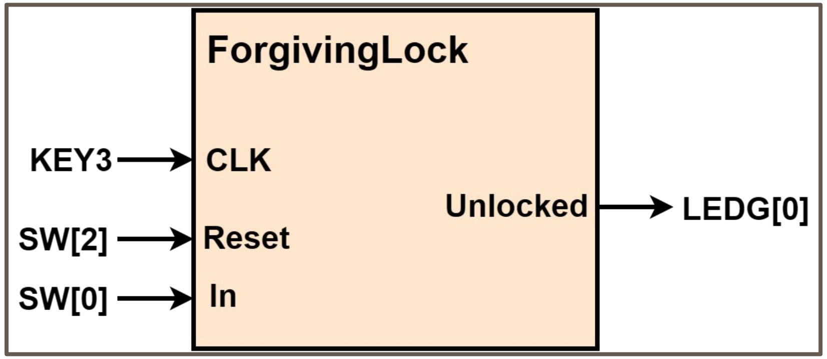
```
module ForgivingLock(
  // TODO: What are the inputs/outputs?
);

  // TODO: What registers/wires should we declare?

  // TODO: Should we have any initialization?

  // TODO: Next State Logic (Combinational or Sequential?)

  // TODO: Memory Logic (Combinational or Sequential?)

  // TODO: Output Logic (Combinational or Sequential?)

endmodule
```

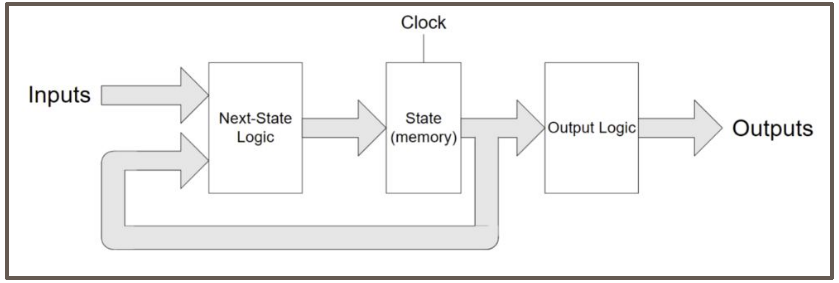


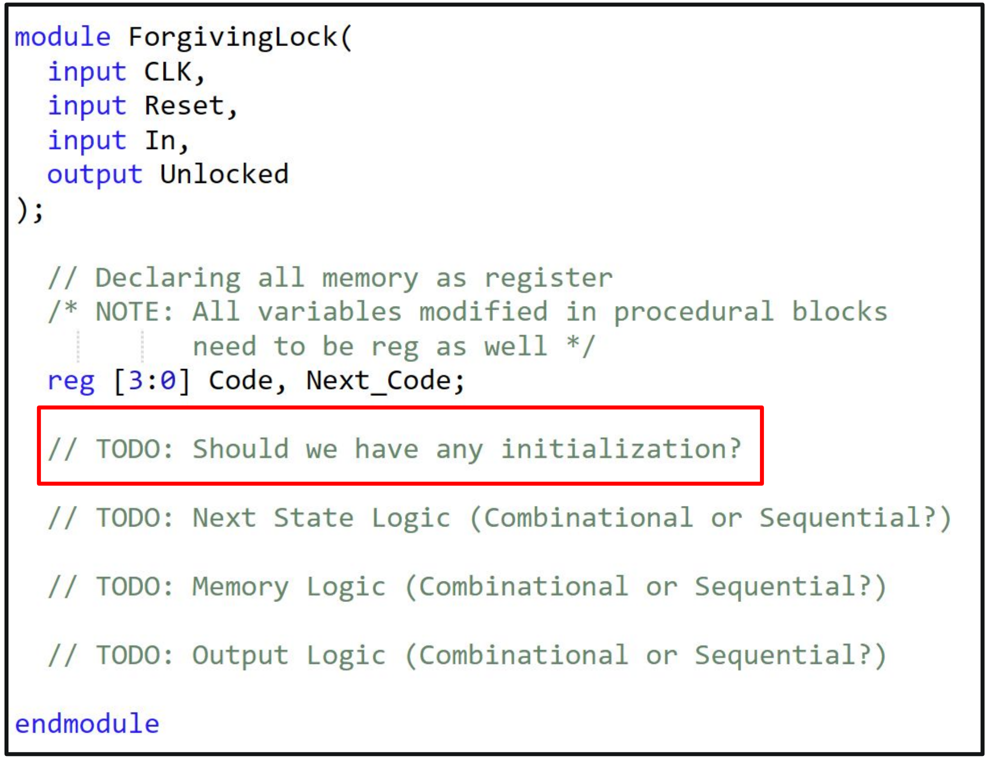
```verilog
module ForgivingLock(
  input CLK,
  input Reset,
  input In,
  output Unlocked
);

  // TODO: What registers/wires should we declare?

  // TODO: Should we have any initialization?

  // TODO: Next State Logic (Combinational or Sequential?)

  // TODO: Memory Logic (Combinational or Sequential?)

  // TODO: Output Logic (Combinational or Sequential?)

endmodule
```



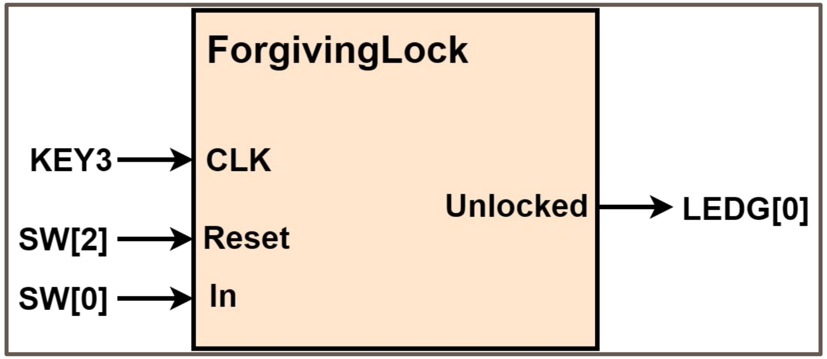
```
module ForgivingLock(
  input CLK,
  input Reset,
  input In,
  output Unlocked
);

  // TODO: What registers/wires should we declare?

  // TODO: Should we have any initialization?

  // TODO: Next State Logic (Combinational or Sequential?)

  // TODO: Memory Logic (Combinational or Sequential?)

  // TODO: Output Logic (Combinational or Sequential?)

endmodule
```

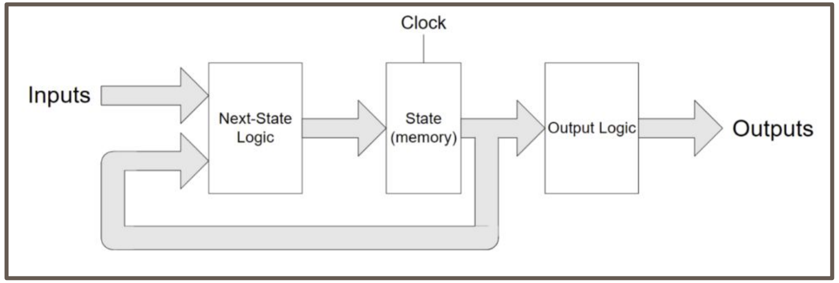




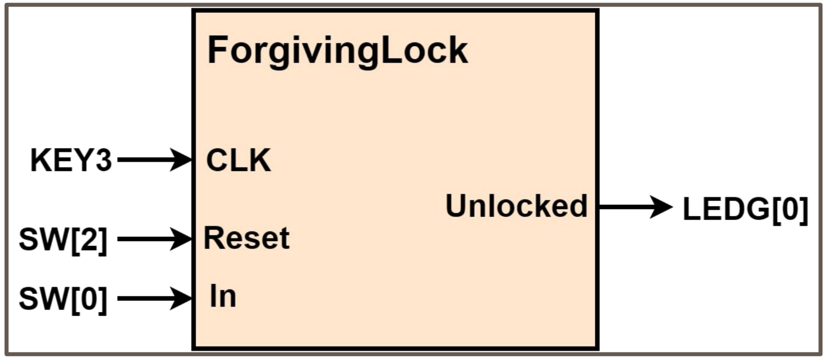


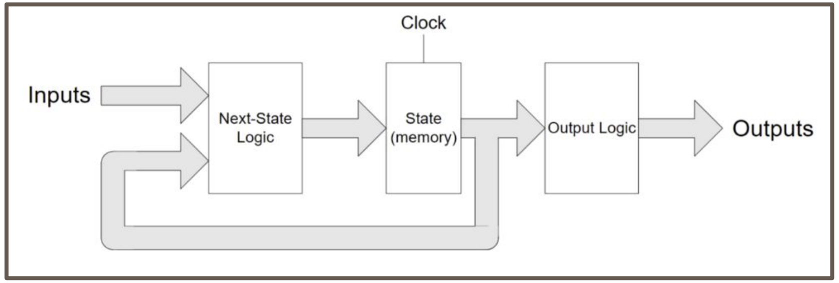
```verilog
module ForgivingLock(
  input CLK,
  input Reset,
  input In,
  output Unlocked
);

  // Declaring all memory as register
  /* NOTE: All variables modified in procedural blocks
           need to be reg as well */
  reg [3:0] Code, Next_Code;

  // TODO: Should we have any initialization?

  // TODO: Next State Logic (Combinational or Sequential?)

  // TODO: Memory Logic (Combinational or Sequential?)

  // TODO: Output Logic (Combinational or Sequential?)

endmodule
```

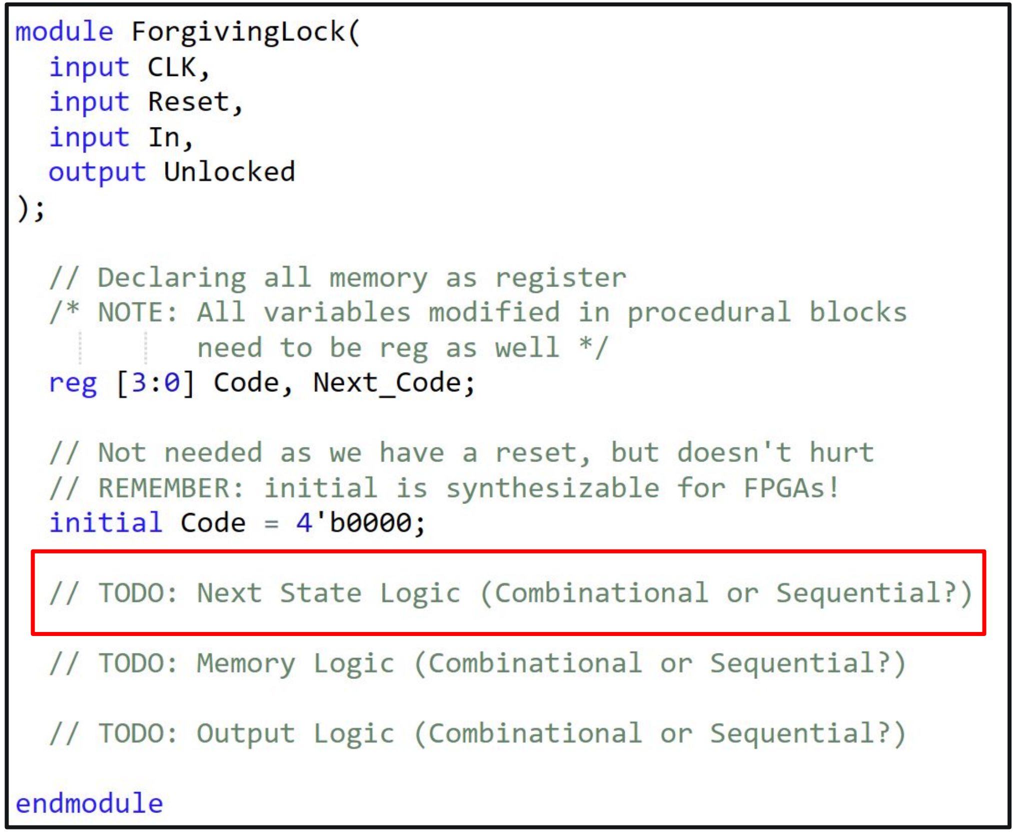




```
module ForgivingLock(
  input CLK,
  input Reset,
  input In,
  output Unlocked
);

  // Declaring all memory as register
  /* NOTE: All variables modified in procedural blocks
           need to be reg as well */
  reg [3:0] Code, Next_Code;

  // Not needed as we have a reset, but doesn't hurt
  // REMEMBER: initial is synthesizable for FPGAs!
  initial Code = 4'b0000;

  // TODO: Next State Logic (Combinational or Sequential?)

  // TODO: Memory Logic (Combinational or Sequential?)

  // TODO: Output Logic (Combinational or Sequential?)

endmodule
```







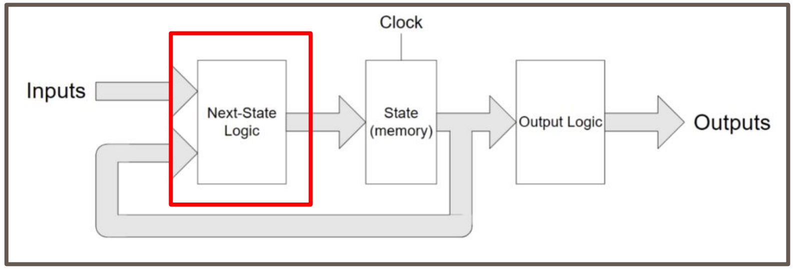
```
module ForgivingLock(
  input CLK,
  input Reset,
  input In,
  output Unlocked
);

  // Declaring all memory as register
  /* NOTE: All variables modified in procedural blocks
           need to be reg as well */
  reg [3:0] Code, Next_Code;

  // Not needed as we have a reset, but doesn't hurt
  // REMEMBER: initial is synthesizable for FPGAs!
  initial Code = 4'b0000;

  // TODO: Next State Logic (Combinational or Sequential?)

  // TODO: Memory Logic (Combinational or Sequential?)

  // TODO: Output Logic (Combinational or Sequential?)

endmodule
```

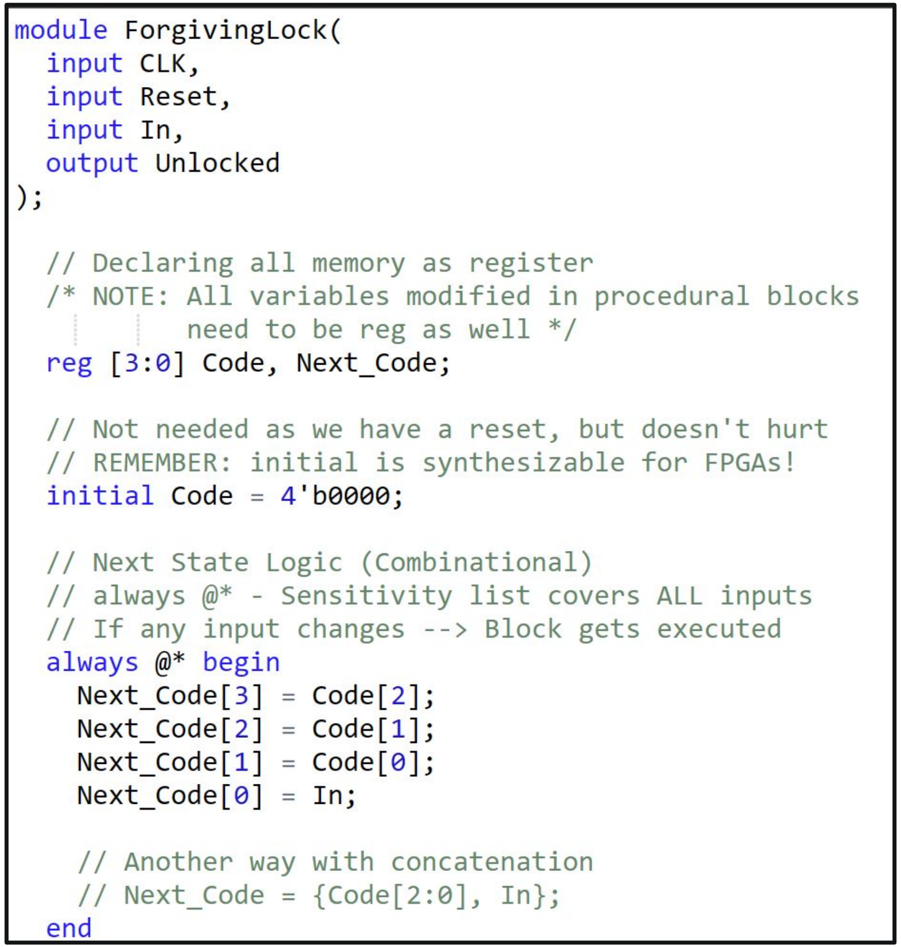
Next_Code[3] = Code[2]

Next_Code[2] = Code[1]

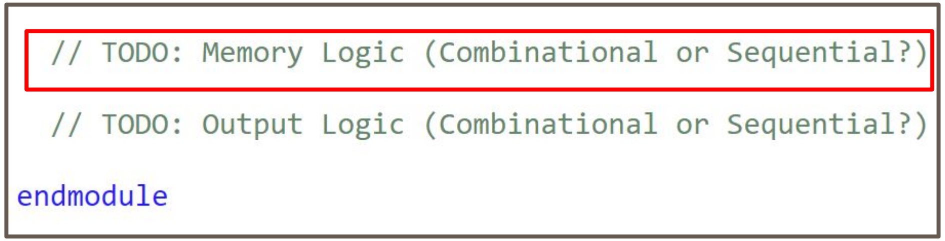
Next_Code[1] = Code[0]

Next_Code[0] = In

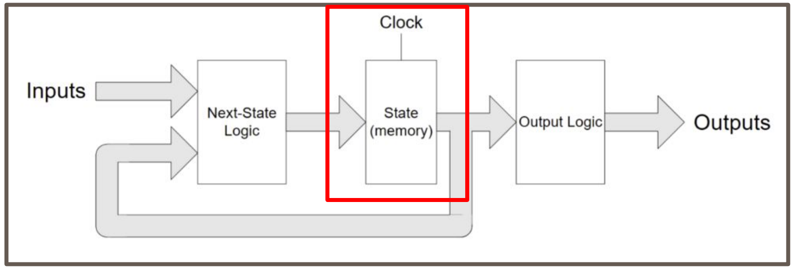


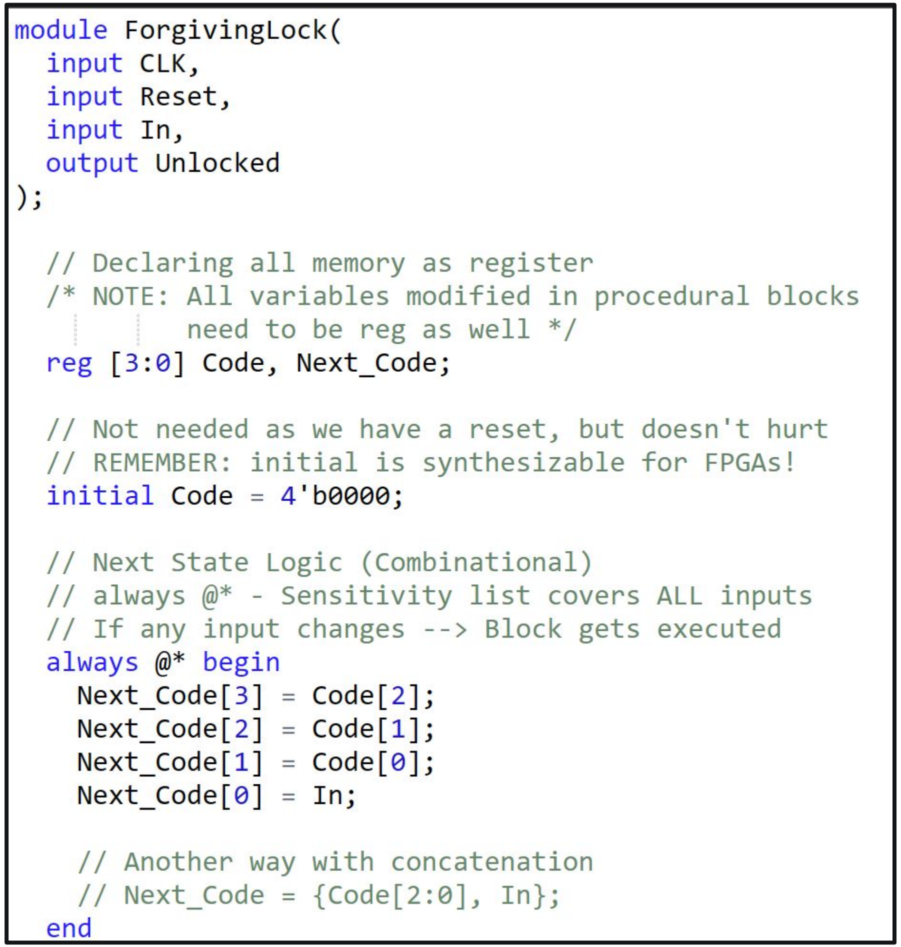


```verilog
module ForgivingLock(
  input CLK,
  input Reset,
  input In,
  output Unlocked
);

  // Declaring all memory as register
  /* NOTE: All variables modified in procedural blocks
           need to be reg as well */
  reg [3:0] Code, Next_Code;

  // Not needed as we have a reset, but doesn't hurt
  // REMEMBER: initial is synthesizable for FPGAs!
  initial Code = 4'b0000;

  // Next State Logic (Combinational)
  // always @* - Sensitivity list covers ALL inputs
  // If any input changes --> Block gets executed
  always @* begin
    Next_Code[3] = Code[2];
    Next_Code[2] = Code[1];
    Next_Code[1] = Code[0];
    Next_Code[0] = In;

    // Another way with concatenation
    // Next_Code = {Code[2:0], In};
  end
```

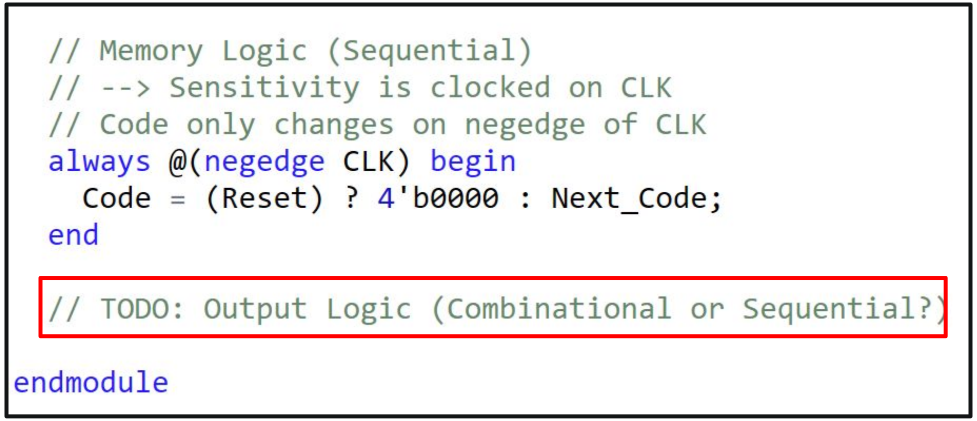


```verilog
  // TODO: Memory Logic (Combinational or Sequential?)

  // TODO: Output Logic (Combinational or Sequential?)

endmodule
```

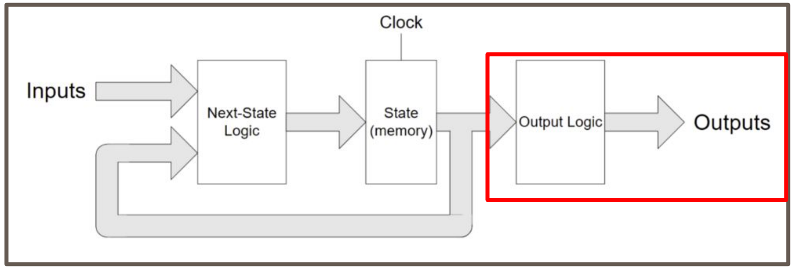




```
module ForgivingLock(
  input CLK,
  input Reset,
  input In,
  output Unlocked
);

  // Declaring all memory as register
  /* NOTE: All variables modified in procedural blocks
           need to be reg as well */
  reg [3:0] Code, Next_Code;

  // Not needed as we have a reset, but doesn't hurt
  // REMEMBER: initial is synthesizable for FPGAs!
  initial Code = 4'b0000;

  // Next State Logic (Combinational)
  // always @* - Sensitivity list covers ALL inputs
  // If any input changes --> Block gets executed
  always @* begin
    Next_Code[3] = Code[2];
    Next_Code[2] = Code[1];
    Next_Code[1] = Code[0];
    Next_Code[0] = In;

    // Another way with concatenation
    // Next_Code = {Code[2:0], In};
  end
```



```
  // Memory Logic (Sequential)
  // --> Sensitivity is clocked on CLK
  // Code only changes on negedge of CLK
  always @(negedge CLK) begin
    Code = (Reset) ? 4'b0000 : Next_Code;
  end

  // TODO: Output Logic (Combinational or Sequential?)

endmodule
```



**Unlocked = (~Code[3] & Code[2] & ~Code[1]) |
(Code[2] & ~Code[1] & Code[0]) |
(Code[3] & ~Code[2] & Code[1]) |
(~Code[2] & Code[1] & ~Code[0])**

```
module ForgivingLock(
  input CLK,
  input Reset,
  input In,
  output Unlocked
);

  // Declaring all memory as register
  /* NOTE: All variables modified in procedural blocks
           need to be reg as well */
  reg [3:0] Code, Next_Code;

  // Not needed as we have a reset, but doesn't hurt
  // REMEMBER: initial is synthesizable for FPGAs!
  initial Code = 4'b0000;

  // Next State Logic (Combinational)
  // always @* - Sensitivity list covers ALL inputs
  // If any input changes --> Block gets executed
  always @* begin
    Next_Code[3] = Code[2];
    Next_Code[2] = Code[1];
    Next_Code[1] = Code[0];
    Next_Code[0] = In;

    // Another way with concatenation
    // Next_Code = {Code[2:0], In};
  end
```

```
  // Memory Logic (Sequential)
  // --> Sensitivity is clocked on CLK
  // Code only changes on negedge of CLK
  always @(negedge CLK) begin
    Code = (Reset) ? 4'b0000 : Next_Code;
  end

  // Output Logic (Combinational)
  assign Unlocked = (~Code[3] & Code[2] & ~Code[1]) |
                    (Code[2] & ~Code[1] & Code[0])  |
                    (Code[3] & ~Code[2] & Code[1])  |
                    (~Code[2] & Code[1] & ~Code[0]);

endmodule
```

(Optional) Try to simply the Unlocked equation above using your theorems!

Try to further simply using XORs!

Final Code for Forgiving Lock

```
`timescale 1ns/1ns

module ForgivingTopLevel(
  input [3:3] KEY,
  input [1:0] SW,
  output [0:0] LEDG
);

  ForgivingLock i(.CLK(KEY[3]), .Reset(SW[1]), .In(SW[0]), .Unlocked(LEDG[0]));

endmodule
```

```
module ForgivingTestbench;

  reg [3:3] KEY;
  reg [1:0] SW;
  wire [0:0] LEDG;

  ForgivingLock dut(.CLK(KEY[3]), .Reset(SW[1]), .In(SW[0]), .Unlocked(LEDG[0]));

  reg correct_LEDG;

  // Task for inserting 0
  task Resetting_Task;
    begin
      SW = 2'b10;
      #5;
      KEY[3] = 1'b0;
      correct_LEDG = 1'b0;
      #5;
      KEY[3] = 1'b1;
    end
  endtask

  // Task for inserting 0
  task Inserting_0_Task;
    input correct_LEDG_val;

    begin
      SW = 2'b00;
      #5;
      KEY[3] = 1'b0;
      correct_LEDG = correct_LEDG_val;
      #5;
      KEY[3] = 1'b1;
    end
  endtask

  // Task for inserting 1
  task Inserting_1_Task;
    input correct_LEDG_val;

    begin
      SW = 2'b01;
      #5;
      KEY[3] = 1'b0;
      correct_LEDG = correct_LEDG_val;
      #5;
      KEY[3] = 1'b1;
    end
  endtask

  initial begin

    // Resetting
    KEY[3] = 1'b1;
    correct_LEDG = 1'b0;
    #5;
    Resetting_Task;

    // Inserting 0
    Inserting_0_Task(1'b0);

    // Cycling through all of the states
    Inserting_1_Task(1'b0);  // 0001
    Inserting_0_Task(1'b1);  // 0010
    Inserting_1_Task(1'b1);  // 0101
    Inserting_0_Task(1'b1);  // 1010
    Inserting_0_Task(1'b1);  // 0100
    Inserting_1_Task(1'b0);  // 1001
    Inserting_1_Task(1'b0);  // 0011
    Inserting_0_Task(1'b0);  // 0110
    Inserting_1_Task(1'b1);  // 1101
    Inserting_1_Task(1'b1);  // 1011
    Inserting_0_Task(1'b0);  // 0110
    Inserting_0_Task(1'b0);  // 1100
    Inserting_0_Task(1'b0);  // 1000
    Inserting_1_Task(1'b0);  // 0001
    Inserting_1_Task(1'b0);  // 0011
    Inserting_1_Task(1'b0);  // 0111
    Inserting_0_Task(1'b0);  // 1110
    Inserting_1_Task(1'b1);  // 1101
    Inserting_1_Task(1'b1);  // 1011
    Inserting_1_Task(1'b0);  // 0111
    Inserting_1_Task(1'b0);  // 1111

  end

endmodule
```

```
module ForgivingLock(
  input CLK,
  input Reset,
  input In,
  output Unlocked
);

  // Declaring all memory as register
  /* NOTE: All variables modified in procedural blocks
          need to be reg as well */
  reg [3:0] Code, Next_Code;

  // Not needed as we have a reset, but doesn't hurt
  // REMEMBER: initial is synthesizable for FPGAs!
  initial Code = 4'b0000;

  // Next State Logic (Combinational)
  // always @* - Sensitivity list covers ALL inputs
  // If any input changes --> Block gets executed
  always @* begin
    Next_Code[3] = Code[2];
    Next_Code[2] = Code[1];
    Next_Code[1] = Code[0];
    Next_Code[0] = In;

    // Another way with concatenation
    // Next_Code = {Code[2:0], In};
  end

  // Memory Logic (Sequential)
  // --> Sensitivity is clocked on CLK
  // Code only changes on negedge of CLK
  always @(negedge CLK) begin
    Code = (Reset) ? 4'b0000 : Next_Code;
  end

  // Output Logic (Combinational)
  assign Unlocked = (~Code[3] & Code[2] & ~Code[1]) |
                   (Code[2] & ~Code[1] & Code[0])  |
                   (Code[3] & ~Code[2] & Code[1])  |
                   (~Code[2] & Code[1] & ~Code[0]);

  // Another way to write after theorem simplifications
  // assign Unlocked = (Code[2] & ~Code[1] & (~Code[3] | Code[0])) |
  //                  (~Code[2] & Code[1] & (Code[3] | ~Code[0]));

  // Going beyond with conceptual and using XOR statements
  // assign Unlocked = ((Code[3] ^ Code[2]) & (Code[2] ^ Code[1])) |
  //                  ((Code[2] ^ Code[1]) & (Code[1] ^ Code[0]));

endmodule
```

Sequential Verilog Practice II (EXTRA-Forgiving Lock)

EXTRA-Forgiving Lock Motivation

- **ISSUE:** I am STILL struggling with typing a correct password.
- **SOLUTION:** Let's give the user some HINTS!

EXTRA-Forgiving Lock Design Problem (Hint #1)

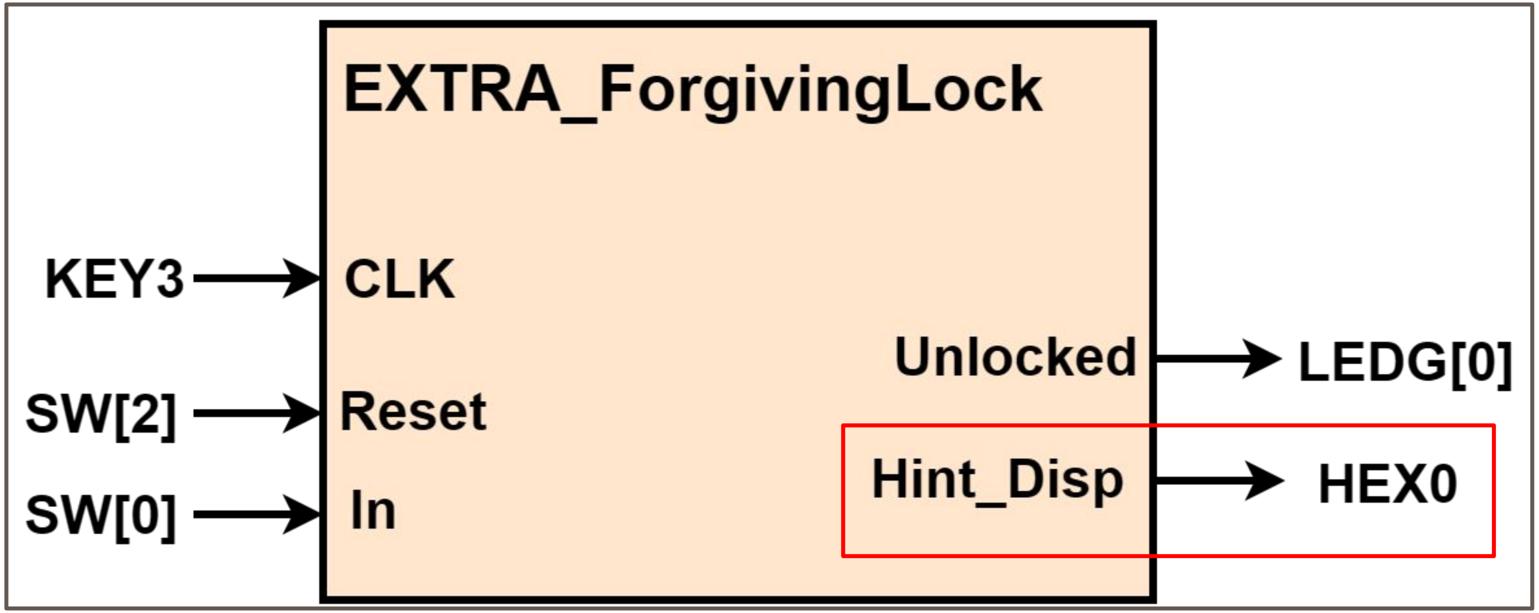
- **Hint #1 (Password Auto-Fill):** We will inform the user what the next digit should be to get closer to a correct password

- Using the same password constraint from before
  - Allow user to unlock if the current code has alternating numbers in at least 3 digits
  - Ex. 010x, 101x, x010, …

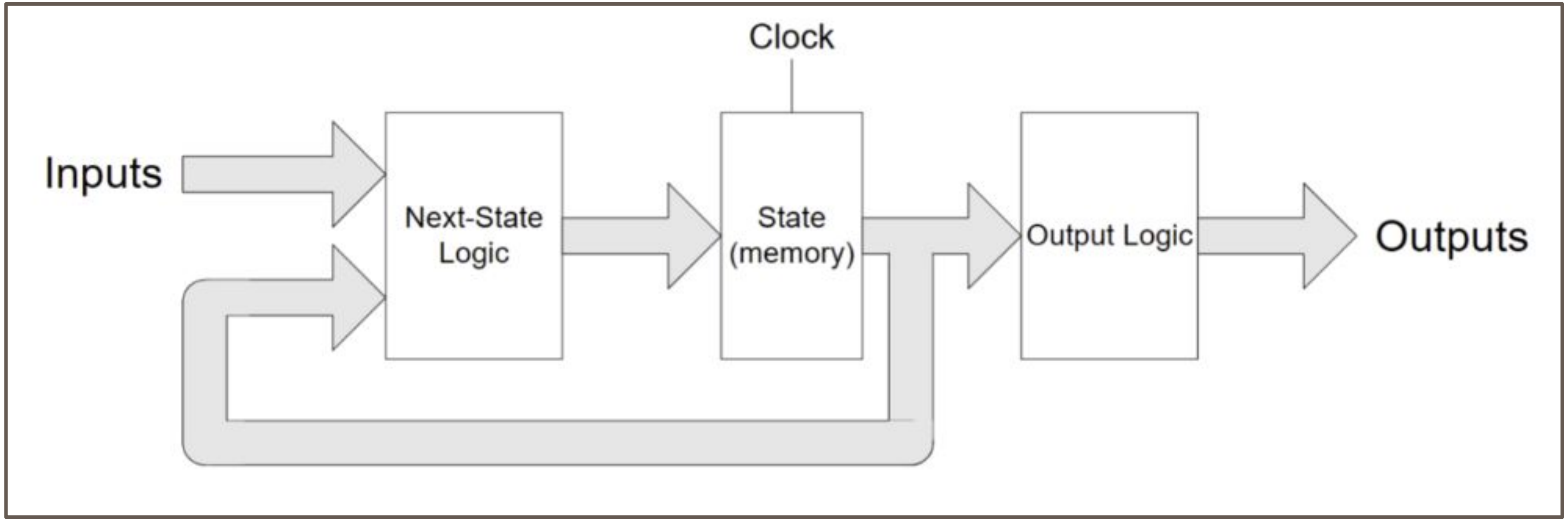
- **NEW: Will display the number the user should provide to get towards a correct password**
  - Ex. Current Stream 1110 → Display ‘1’
  - Ex. Current Stream 1001 → Display ‘0’
  - Ex. Current Stream 1010 → Display ‘1’
  - Ex. Current Stream 1111 → Display ‘0’

EXTRA-Forgiving Lock Design Diagram

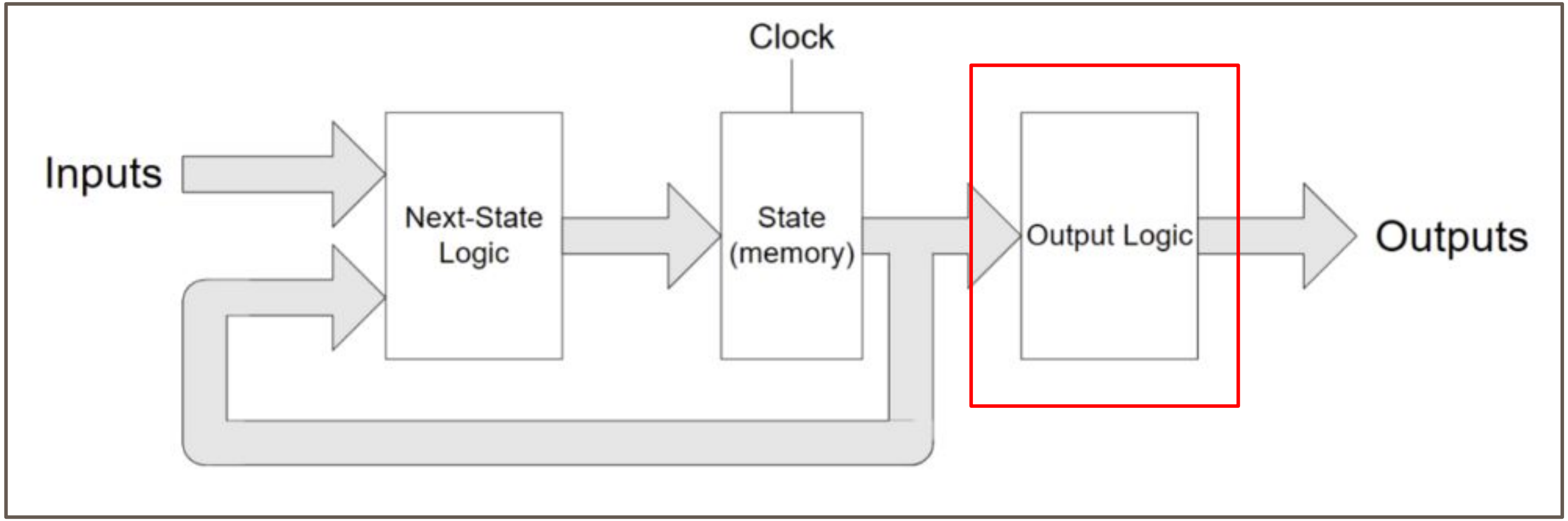
- Addition of a HEX display output



What’s Going to Change?



What’s Going to Change?



EXTRA-Forgiving Lock Output Table / Equation

Code	Hint_Disp
0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	

Code	Hint_Disp
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

Hint_Disp = ?

EXTRA-Forgiving Lock Output Table / Equation

Code	Hint_Disp
0000	“1”
0001	
0010	
0011	
0100	
0101	
0110	
0111	

Code	Hint_Disp
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

Hint_Disp = ?

EXTRA-Forgiving Lock Output Table / Equation

Code	Hint_Disp
0000	“1”
0001	“0”
0010	
0011	
0100	
0101	
0110	
0111	

Code	Hint_Disp
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

Hint_Disp = ?

EXTRA-Forgiving Lock Output Table / Equation

Code	Hint_Disp
0000	“1”
0001	“0”
0010	“1”
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

Hint_Disp = ?

EXTRA-Forgiving Lock Output Table / Equation

Code	Hint_Disp
0000	“1”
0001	“0”
0010	“1”
0011	“0”
0100	“1”
0101	“0”
0110	“1”
0111	“0”

Code	Hint_Disp
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

Hint_Disp = ?

EXTRA-Forgiving Lock Output Table / Equation

Code	Hint_Disp
0000	“1”
0001	“0”
0010	“1”
0011	“0”
0100	“1”
0101	“0”
0110	“1”
0111	“0”

Code	Hint_Disp
1000	“1”
1001	“0”
1010	“1”
1011	“0”
1100	“1”
1101	“0”
1110	“1”
1111	“0”

Hint_Disp = ?

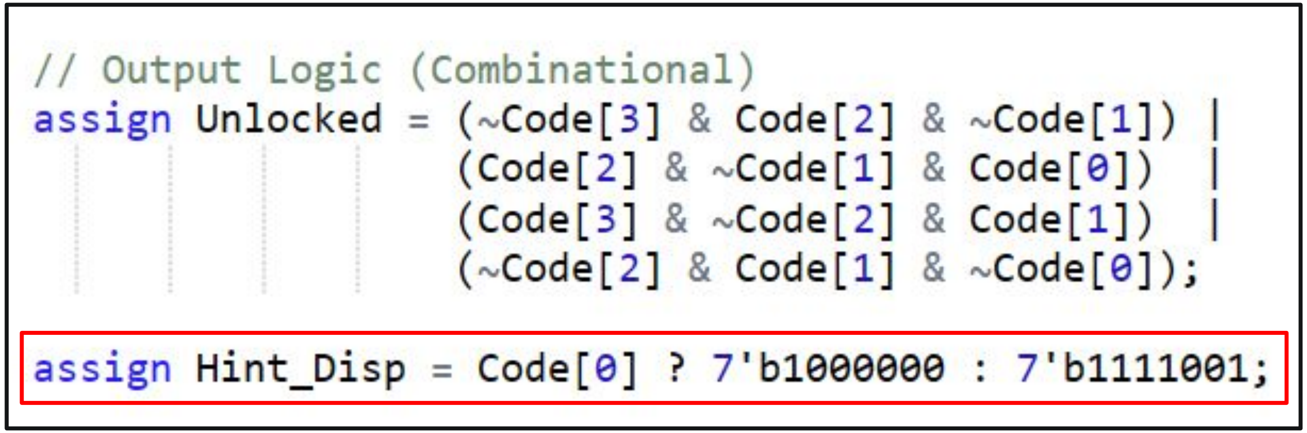
EXTRA-Forgiving Lock Output Table / Equation

Code	Hint_Disp
0000	“1”
0001	“0”
0010	“1”
0011	“0”
0100	“1”
0101	“0”
0110	“1”
0111	“0”

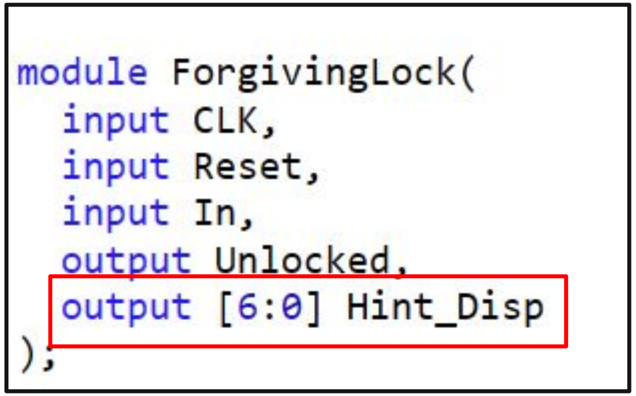
Code	Hint_Disp
1000	“1”
1001	“0”
1010	“1”
1011	“0”
1100	“1”
1101	“0”
1110	“1”
1111	“0”

Hint_Disp =
Code[0] ? “0” : “1”

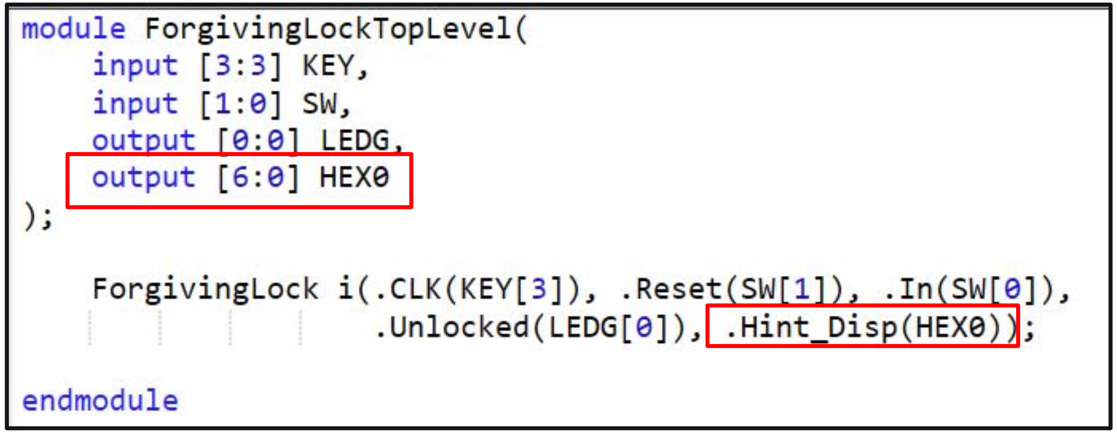
Small Changes to Code



```
// Output Logic (Combinational)
assign Unlocked = (~Code[3] & Code[2] & ~Code[1]) |
                  (Code[2] & ~Code[1] & Code[0])  |
                  (Code[3] & ~Code[2] & Code[1])  |
                  (~Code[2] & Code[1] & ~Code[0]);

assign Hint_Disp = Code[0] ? 7'b1000000 : 7'b1111001;
```



```
module ForgivingLock(
  input CLK,
  input Reset,
  input In,
  output Unlocked,
  output [6:0] Hint_Disp
);
```



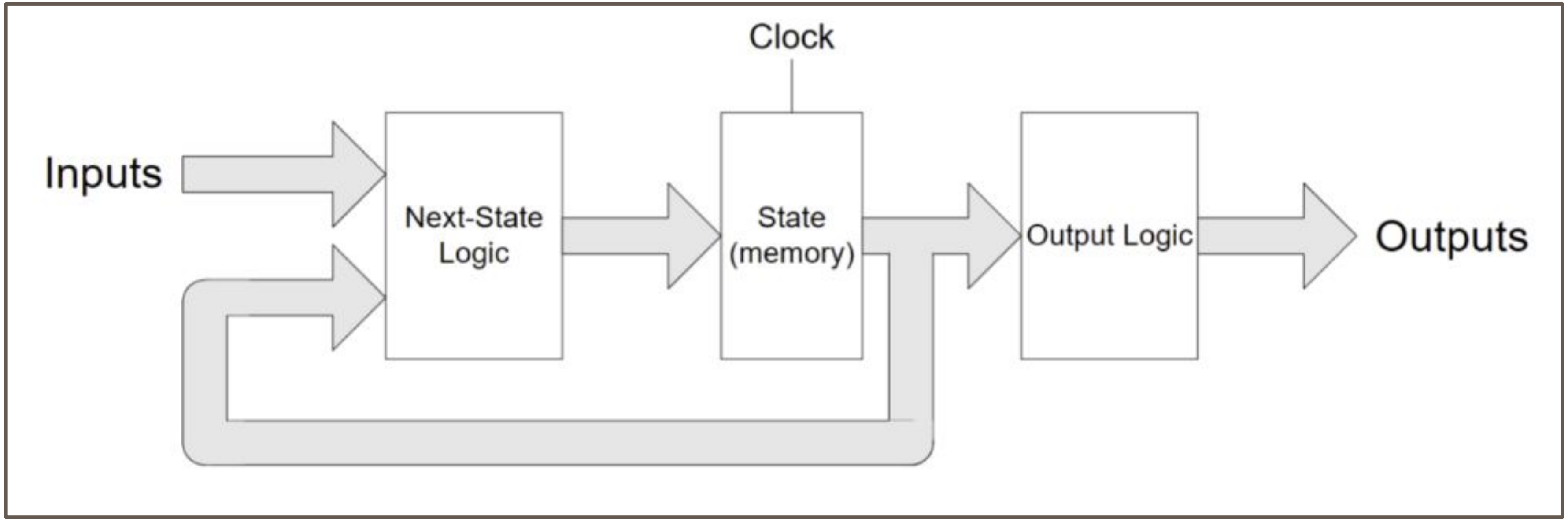
```
module ForgivingLockTopLevel(
    input [3:3] KEY,
    input [1:0] SW,
    output [0:0] LEDG,
    output [6:0] HEX0
);

    ForgivingLock i(.CLK(KEY[3]), .Reset(SW[1]), .In(SW[0]),
                    .Unlocked(LEDG[0]), .Hint_Disp(HEX0));

endmodule
```

EXTRA-Forgiving Lock Design Problem (Hint #2)

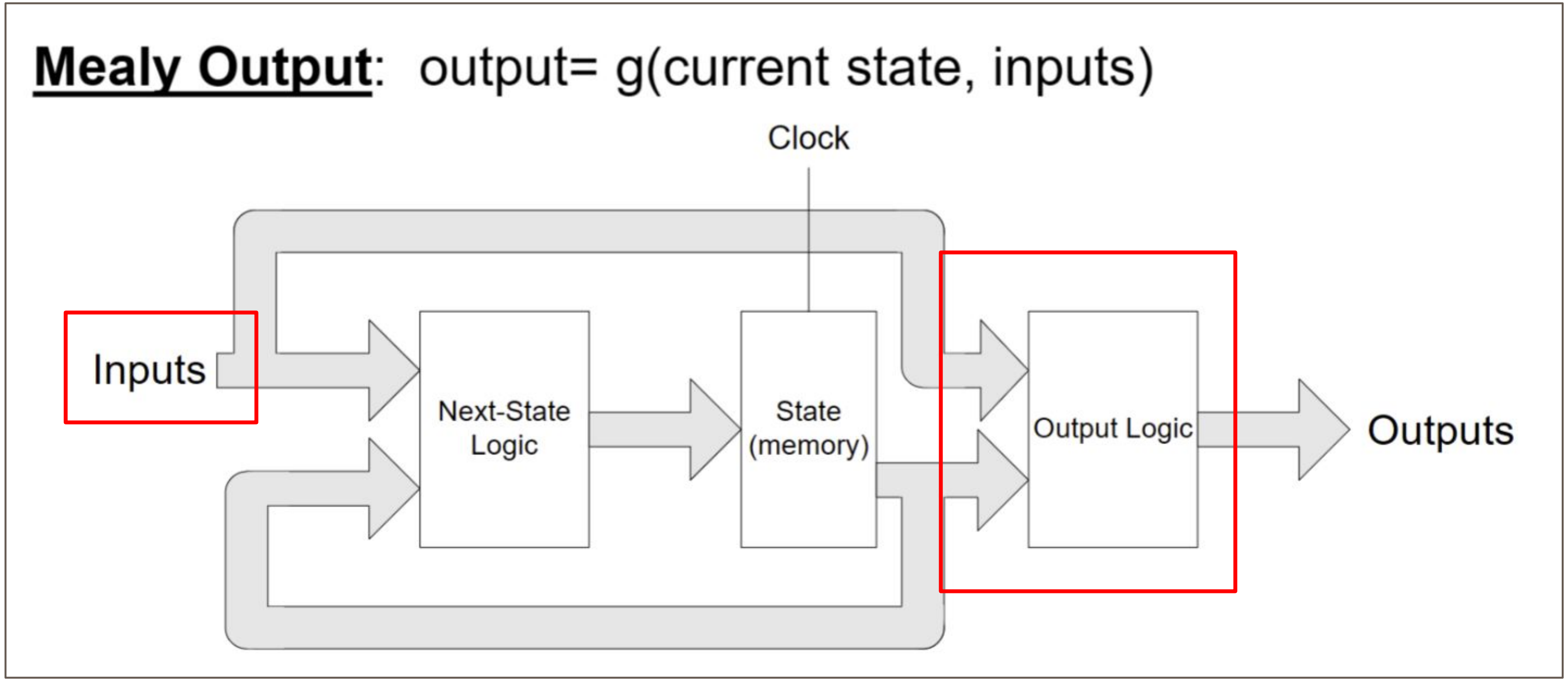
- **Hint #2 (Almost There!):** We want to let the user know if the input they are ABOUT to select will result in a correct password state
  - The user has not yet actually chosen the given input at this time

- **Will turn on an indicator to alert the user that choosing this value will result in a correct password state**
  - Ex. (Current Stream = 1110) & (In = 0) → **Indicator OFF (Will not be correct password)**
  - Ex. (Current Stream = 1001) & (In = 0) → **Indicator ON (Will BECOME correct password)**
  - Ex. (Current Stream = 1010) & (In = 0) → **Indicator ON (Will STAY correct password)**
  - Ex. (Current Stream = 0100) & (In = 1) → **Indicator OFF (Will not be correct password)**

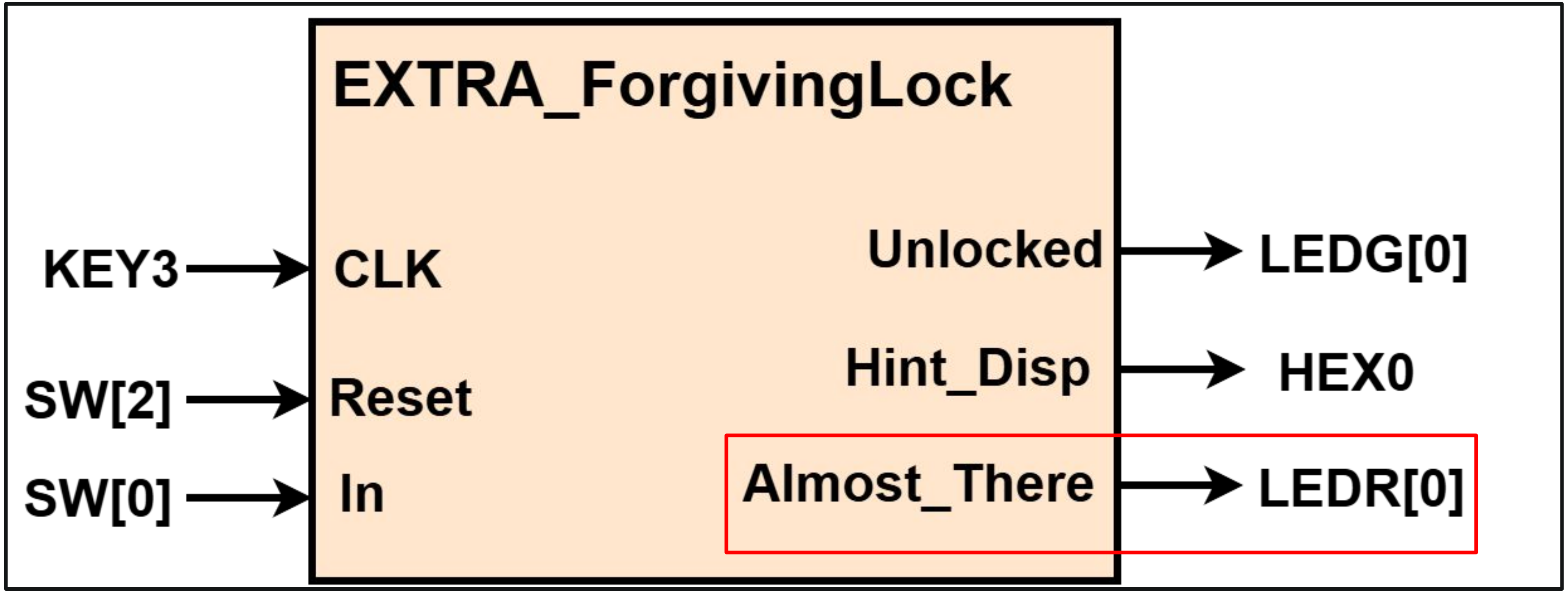
What's Wrong with This Model?



The Indicator is a Mealy Output!!


Mealy Output: output= g(current state, inputs)

EXTRA-Forgiving Lock Design Diagram (Hint #2)

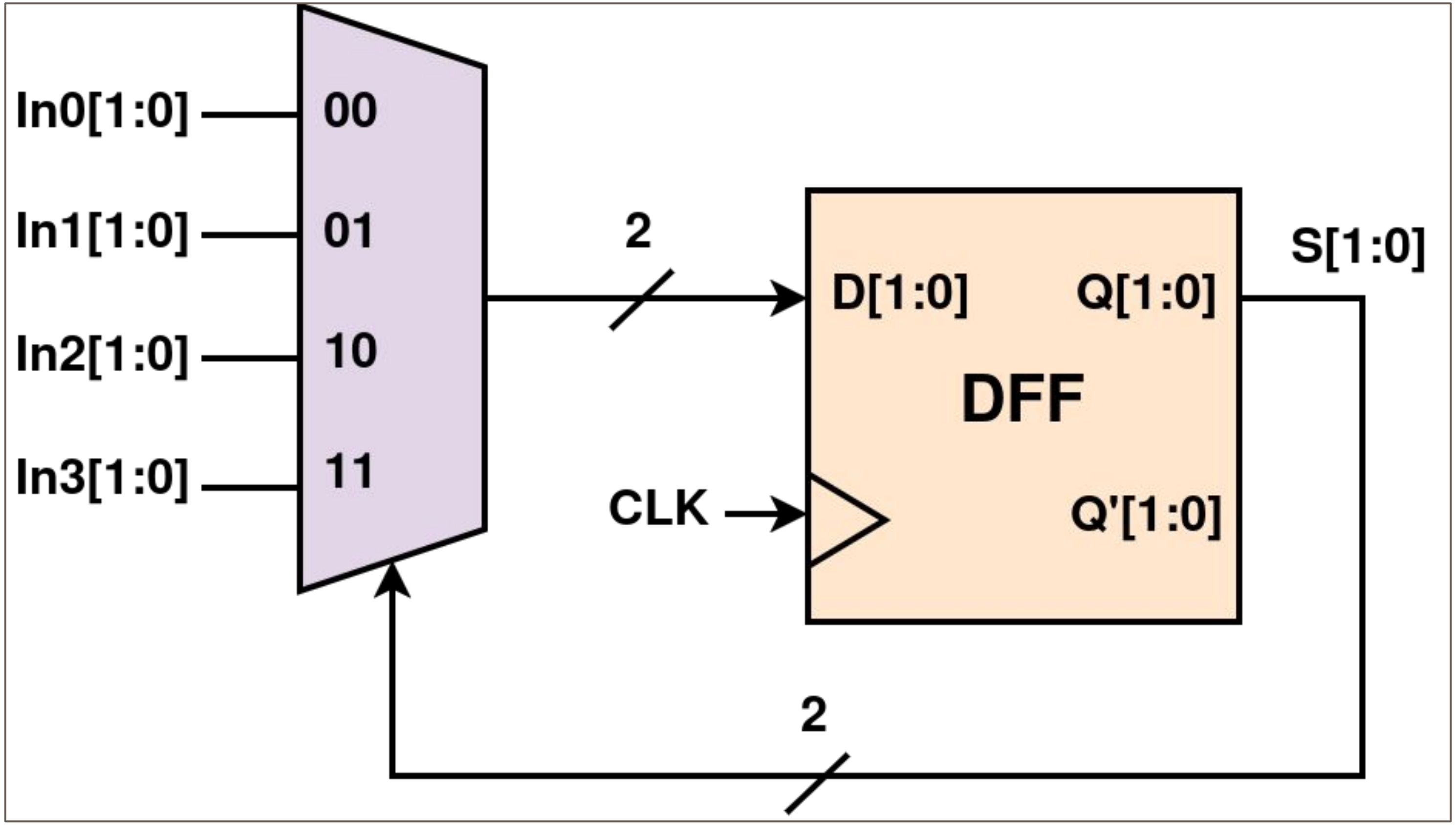


Now it's your turn to update the code!

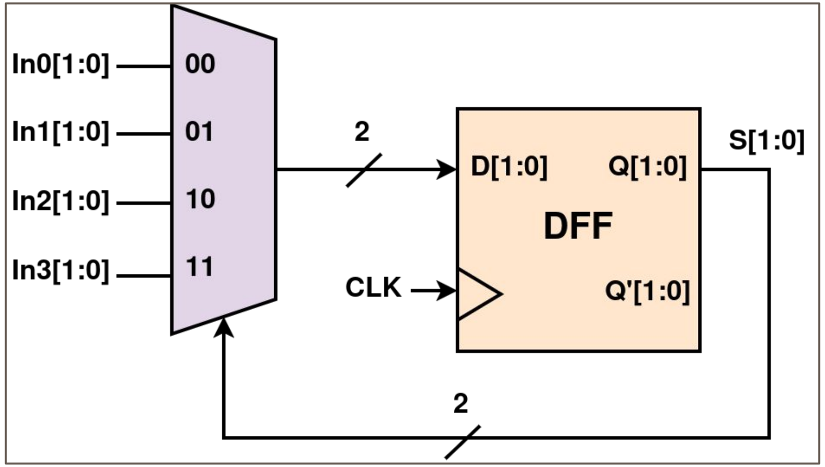
THANK YOU

(BONUS) Sequential Verilog Practice III (Self-Selecting Mux)

Self-Selecting Clocked Mux!!



Some Questions to Ask

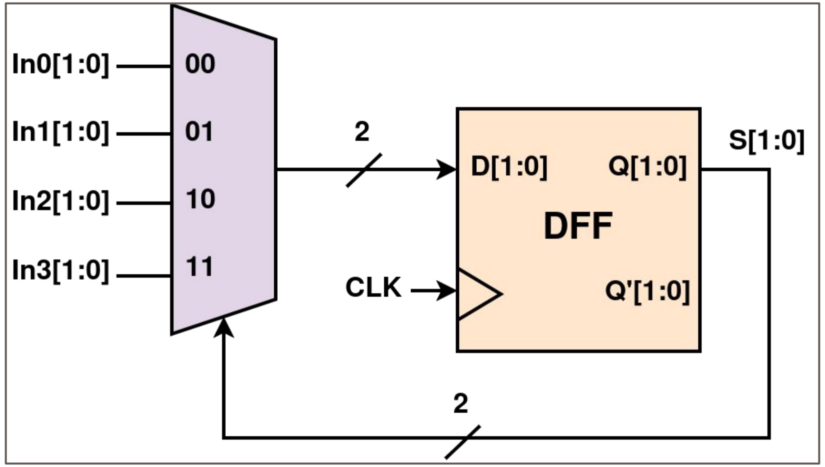


1. **What is a DFF?**
2. **What is CLK?**
3. **What is the diagram even doing?**
4. **How many states exist in this circuit?**
5. **What are those states?**

State Assignments

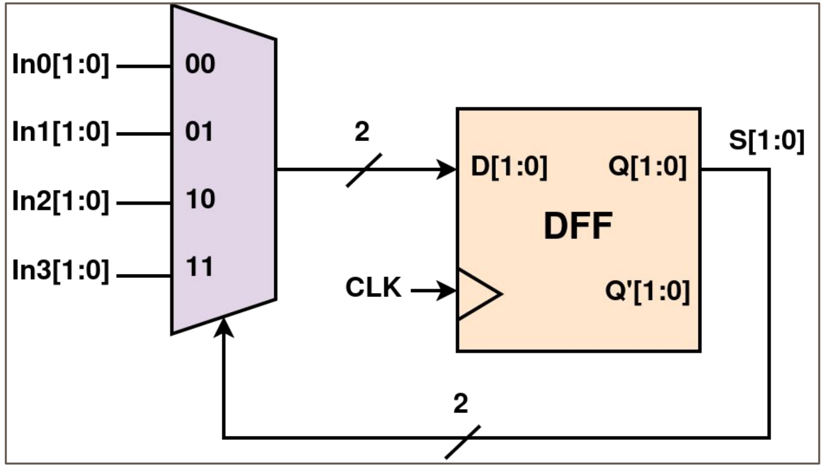
Build a State Assignment Table

State Bits		State Name
?	?	



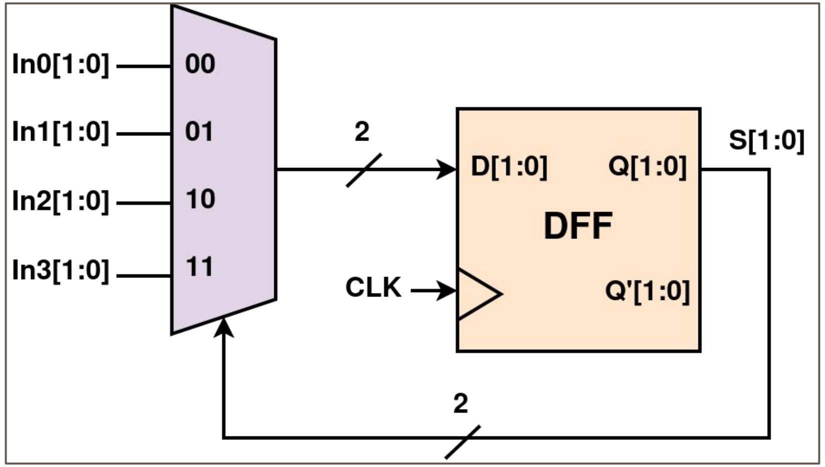
Build a State Assignment Table

State Bits		State Name
S[1]	S[0]	
?	?	
?	?	
?	?	
?	?	



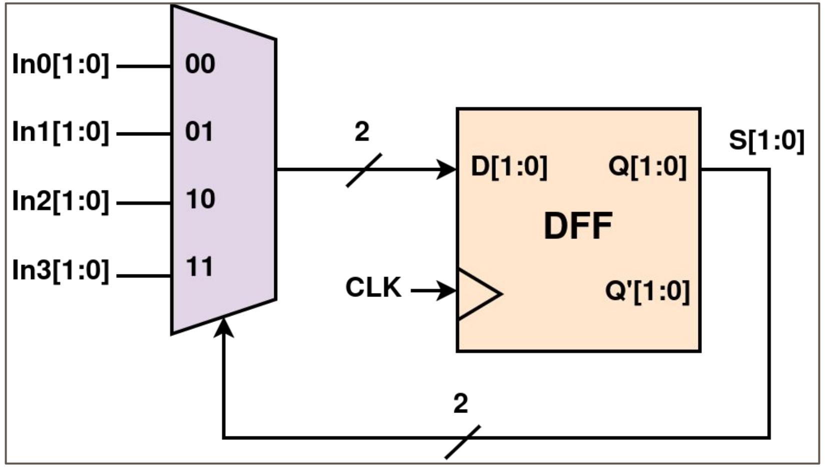
Build a State Assignment Table

State Bits		State Name
S[1]	S[0]	
0	0	?
0	1	?
1	0	?
1	1	?



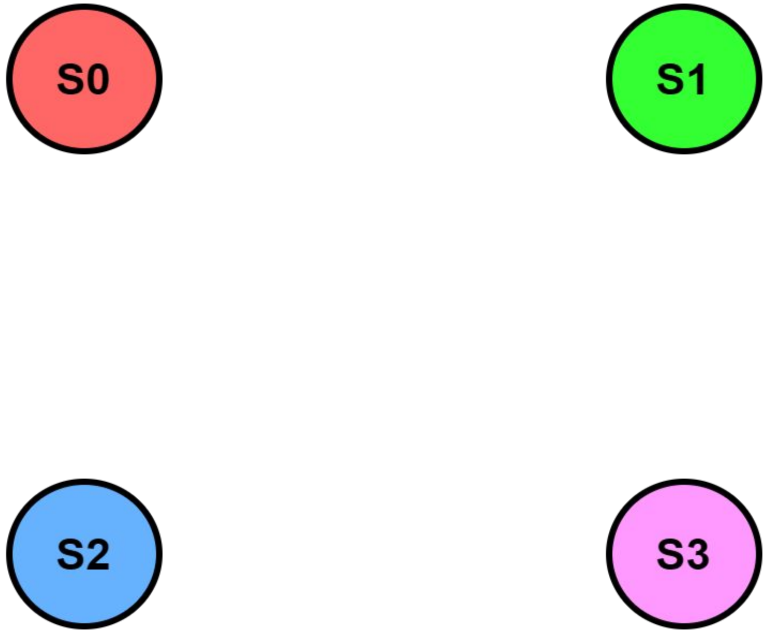
Build a State Assignment Table

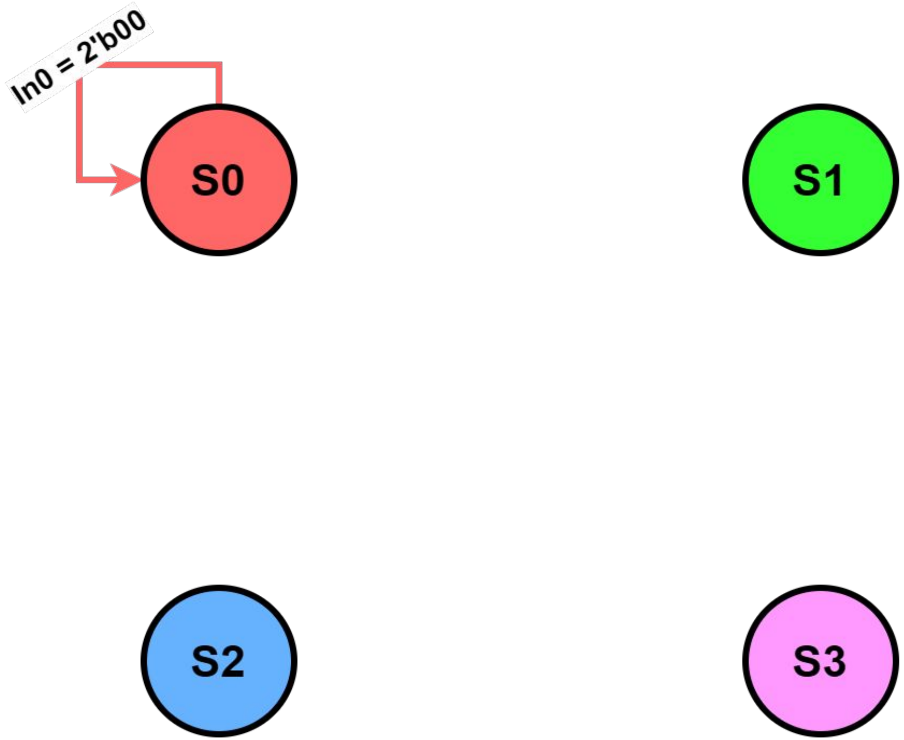
State Bits		State Name
S[1]	S[0]	
0	0	S0
0	1	S1
1	0	S2
1	1	S3

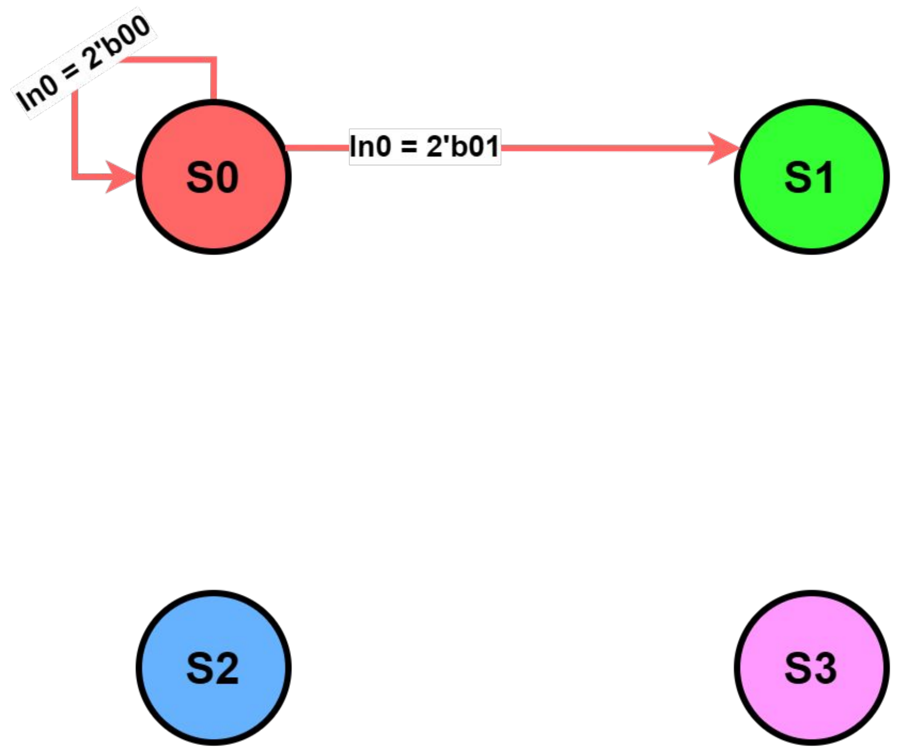


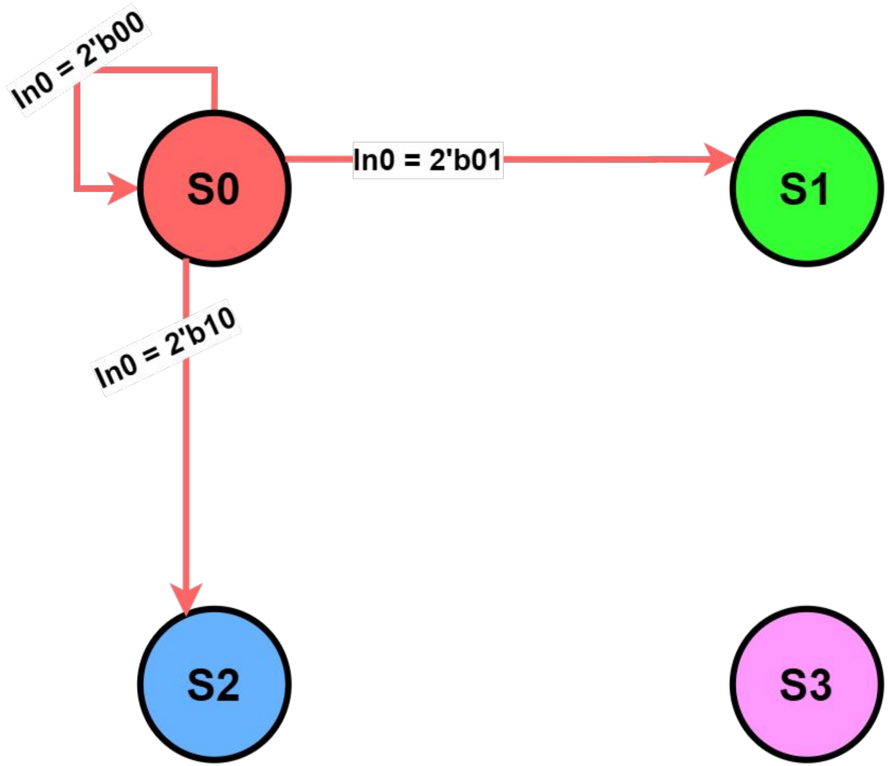
Constructing State Transition Diagram

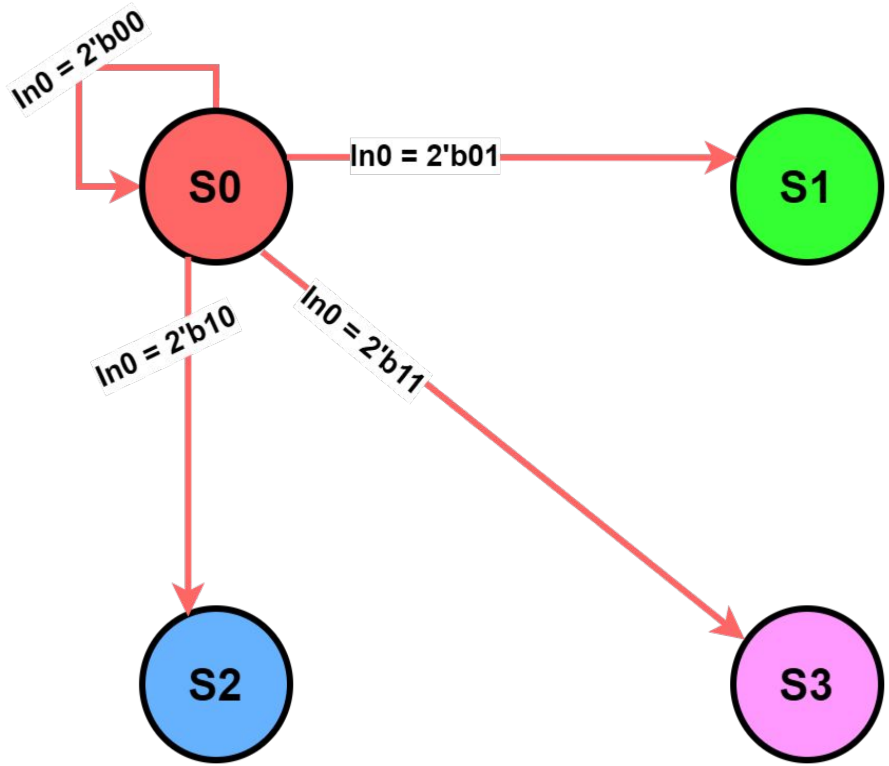
Let’s Build a State Diagram

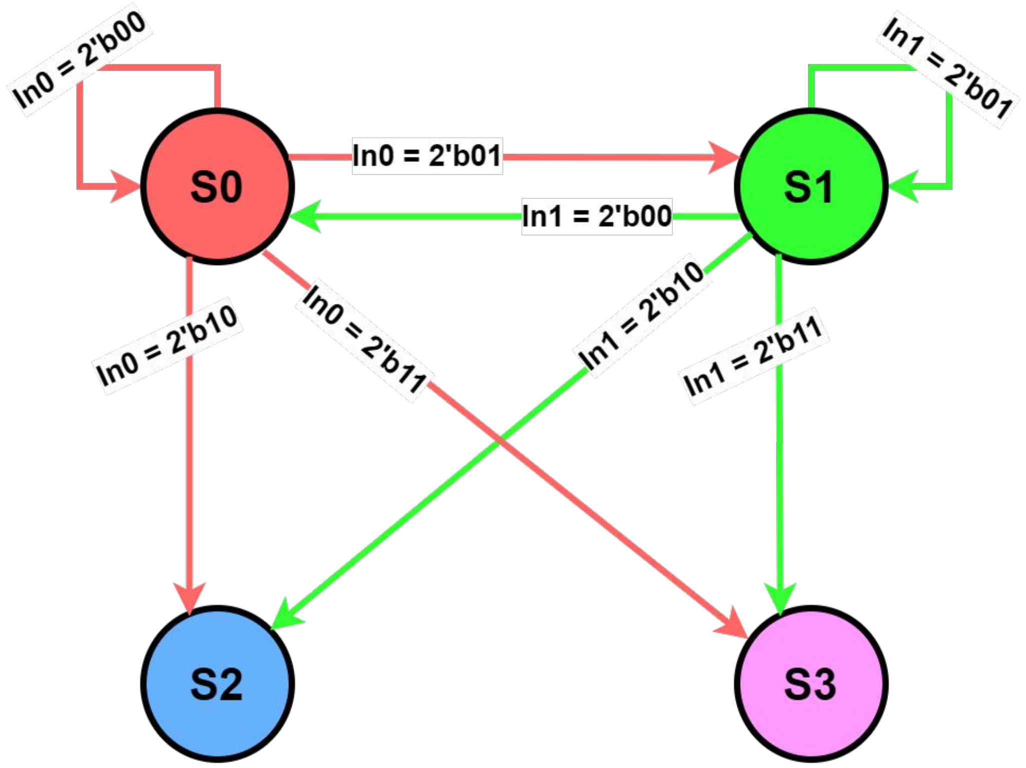


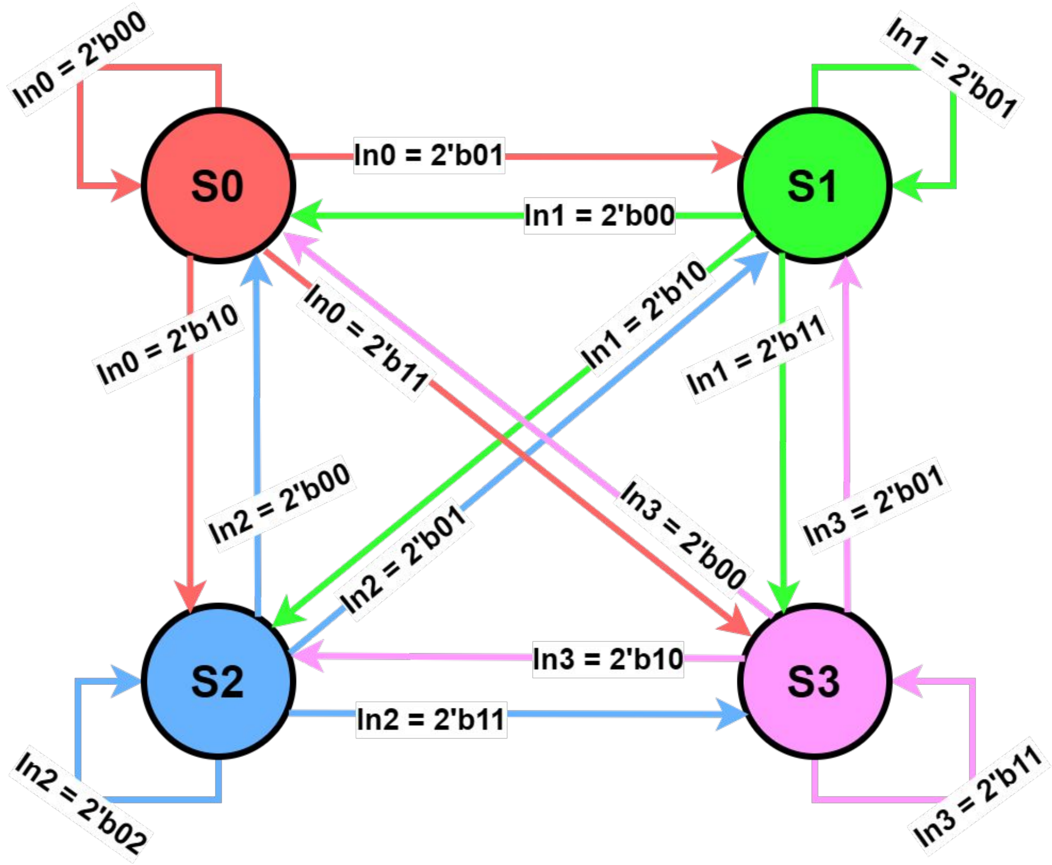

In0 = 2'b00
S0
S1
S2
S3


In0 = 2'b00
In0 = 2'b01
S0
S1
S2
S3


In0 = 2'b00
In0 = 2'b01
S0
S1
In0 = 2'b10
S2
S3


In0 = 2'b00
In0 = 2'b01
In0 = 2'b10
In0 = 2'b11
S0
S1
S2
S3

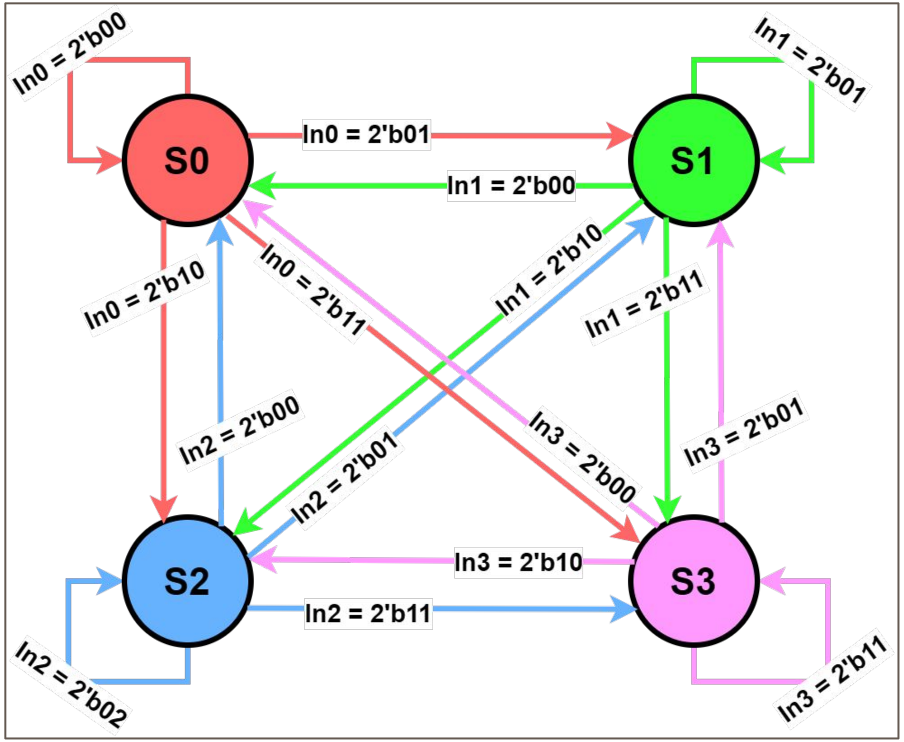

In0 = 2'b00
In1 = 2'b01
S0
In0 = 2'b01
S1
In1 = 2'b00
In0 = 2'b10
In0 = 2'b11
In1 = 2'b10
In1 = 2'b11
S2
S3


S0
S1
S2
S3
In0 = 2'b00
In0 = 2'b01
In0 = 2'b10
In0 = 2'b11
In1 = 2'b01
In1 = 2'b00
In1 = 2'b10
In1 = 2'b11
In2 = 2'b00
In2 = 2'b01
In2 = 2'b11
In2 = 2'b02
In3 = 2'b00
In3 = 2'b01
In3 = 2'b10
In3 = 2'b11

Generating High-Level Next State Equation

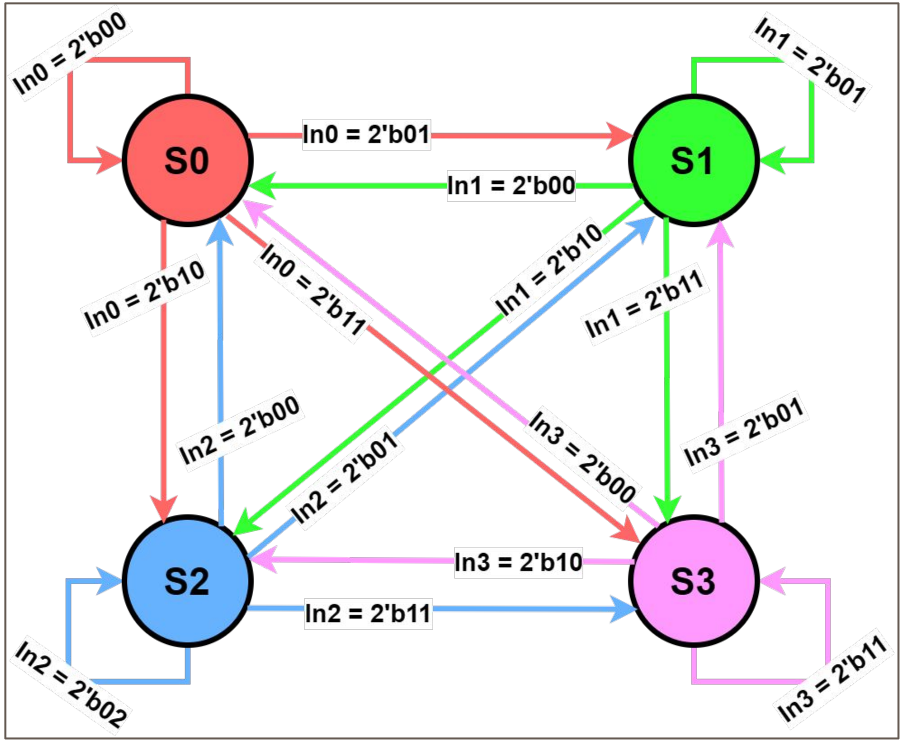
Generating High-Level Next State Equation

```
if (S = ??):
  S⁺ = ??
else if (S = ??):
  S⁺ = ??
else if (S = ??):
  S⁺ = ??
else:
  S⁺ = ??
```



Generating High-Level Next State Equation

```
if (S = S0):
    $S^+$ = In0
else if (S = S1):
    $S^+$ = In1
else if (S = S2):
    $S^+$ = In2
else:
    $S^+$ = In3
```



In0 = 2'b00
In1 = 2'b01
S0
In0 = 2'b01
S1
In1 = 2'b00
In1 = 2'b10
In0 = 2'b10
In0 = 2'b11
In1 = 2'b11
In2 = 2'b00
In2 = 2'b01
In3 = 2'b00
In3 = 2'b01
In3 = 2'b10
S2
S3
In2 = 2'b11
In2 = 2'b02
In3 = 2'b11