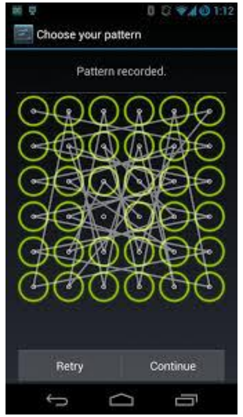
Forgiving Lock Part 1

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

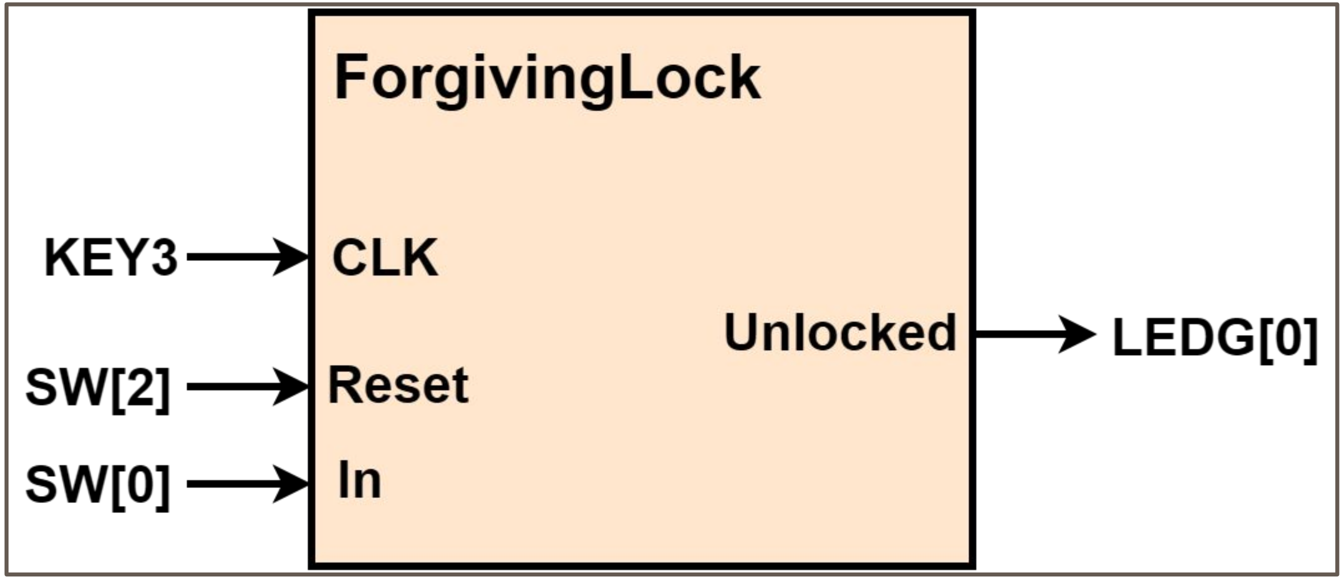
- Allowing users to provide a 4-bit continuous input stream password
  - 0000 –(**1**)→ 000**1** –(**0**)→ 001**0** –(**1**)→ 010**1** –(**1**)→ 101**1** –(**0**)→ 011**0**
  - User must push button to enter new bit into the stream

- 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

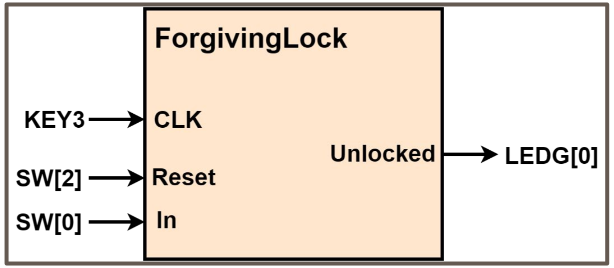
- **Example Loosened Password Constraint**
  - **Allow user to unlock if the current code has alternating numbers in at least 3 digits**
  - **Ex. 010x, 101x, x010, …**

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]



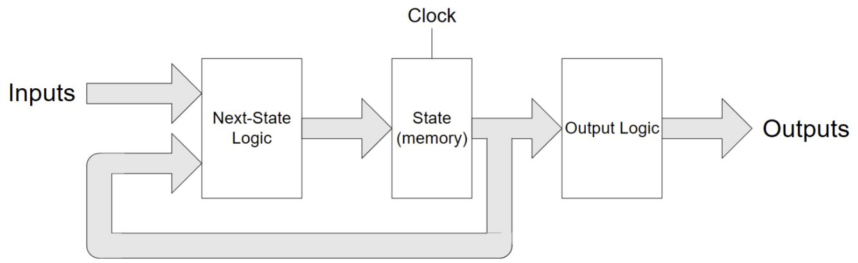
Some Questions



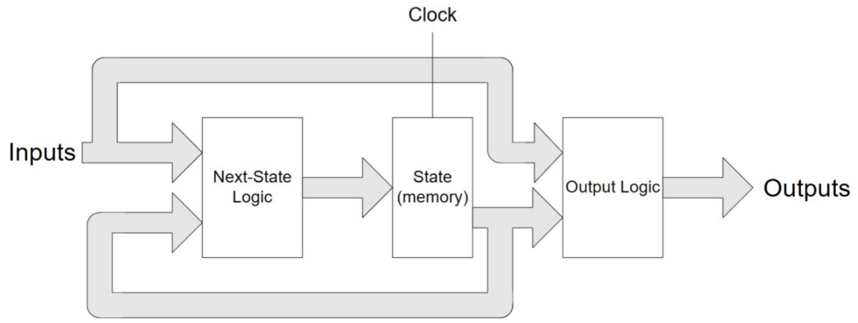
1. **Is this a Moore/Mealy Machine?**
2. **How many internal states exist in this machine?**
3. **For synchronization, should we be using a latch or a FF? Why?**
4. **What does it mean for our machine to be synchronized?**
   a. **If our reset was ‘asynchronous’, what changes?**
   b. **Why might we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

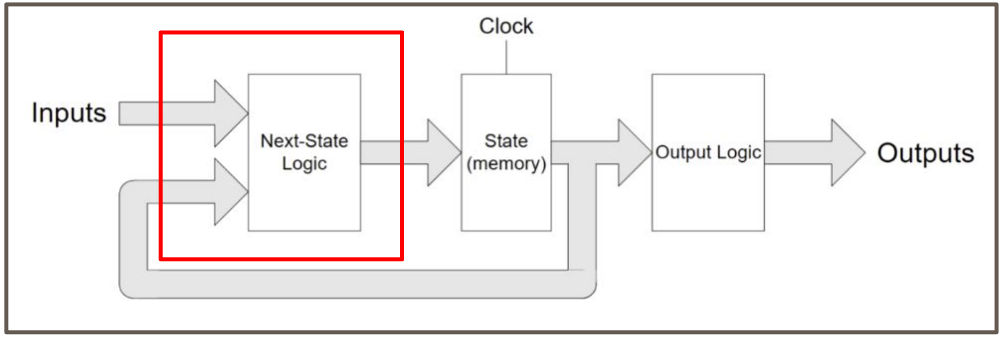
Code	Input (In)		Unlocked
	0	1	
1000	0000	0001	0
1001	0010	0011	0
1010	**0100**	**0101**	**1**
1011	**0110**	**0111**	**1**
1100	1000	1001	0
1101	**1010**	**1011**	**1**
1110	1100	1101	0
1111	1110	1111	0

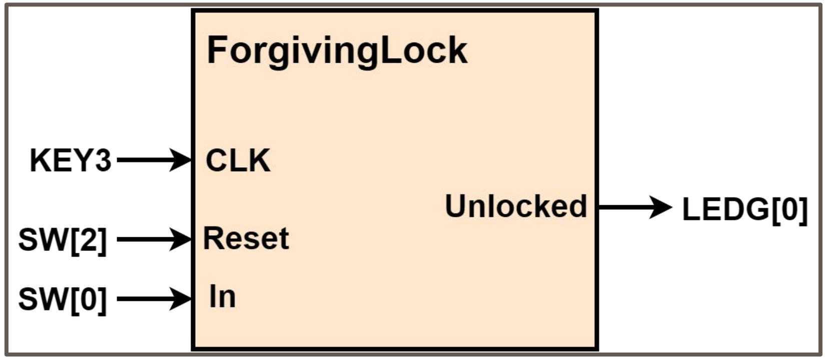
Code^+
(Next_Code)

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





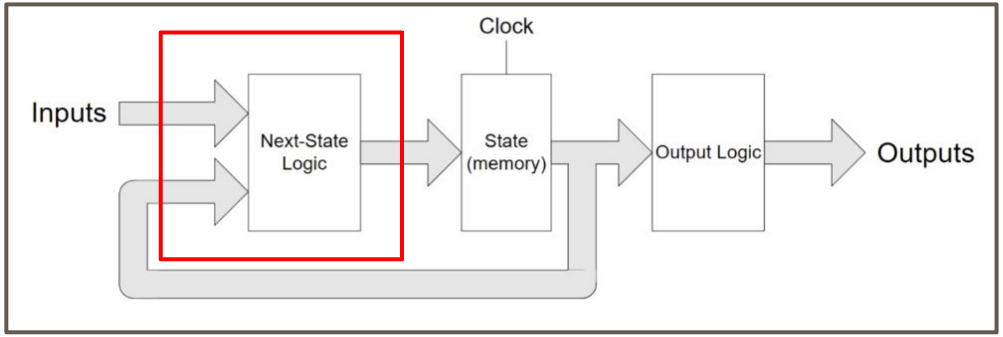
Creating Next State (Transition) Equations

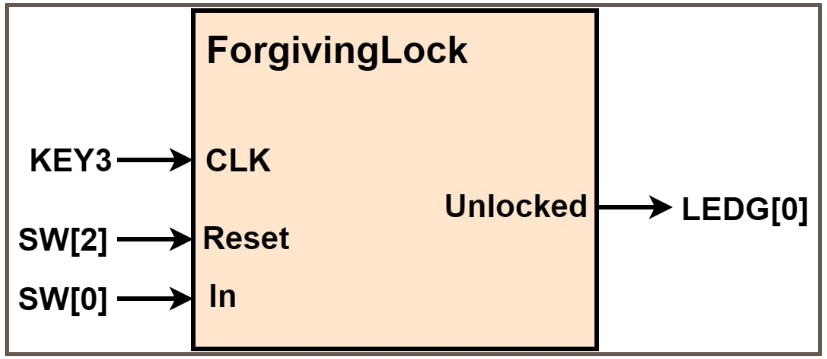
Next_Code[3] = Code[2]

Next_Code[2] = Code[1]

Next_Code[1] = Code[0]

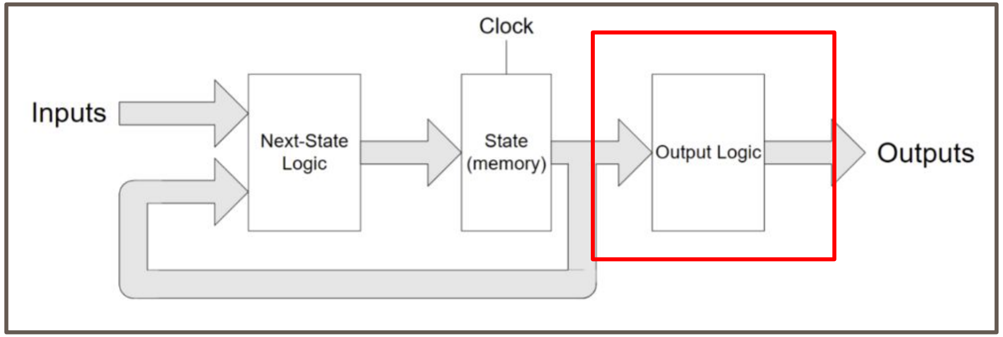
Next_Code[0] = In

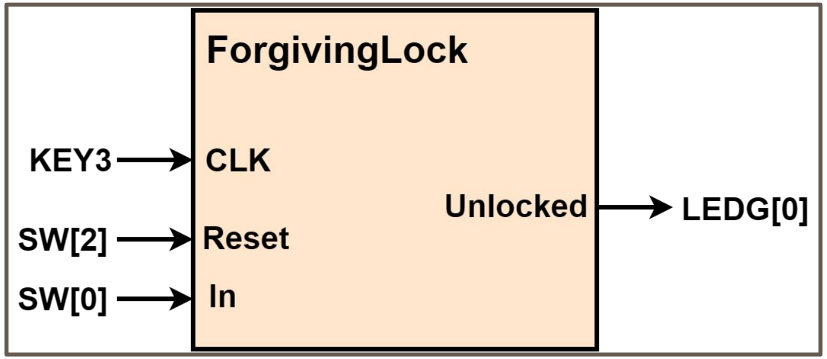




Creating Output Equations

- 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)**

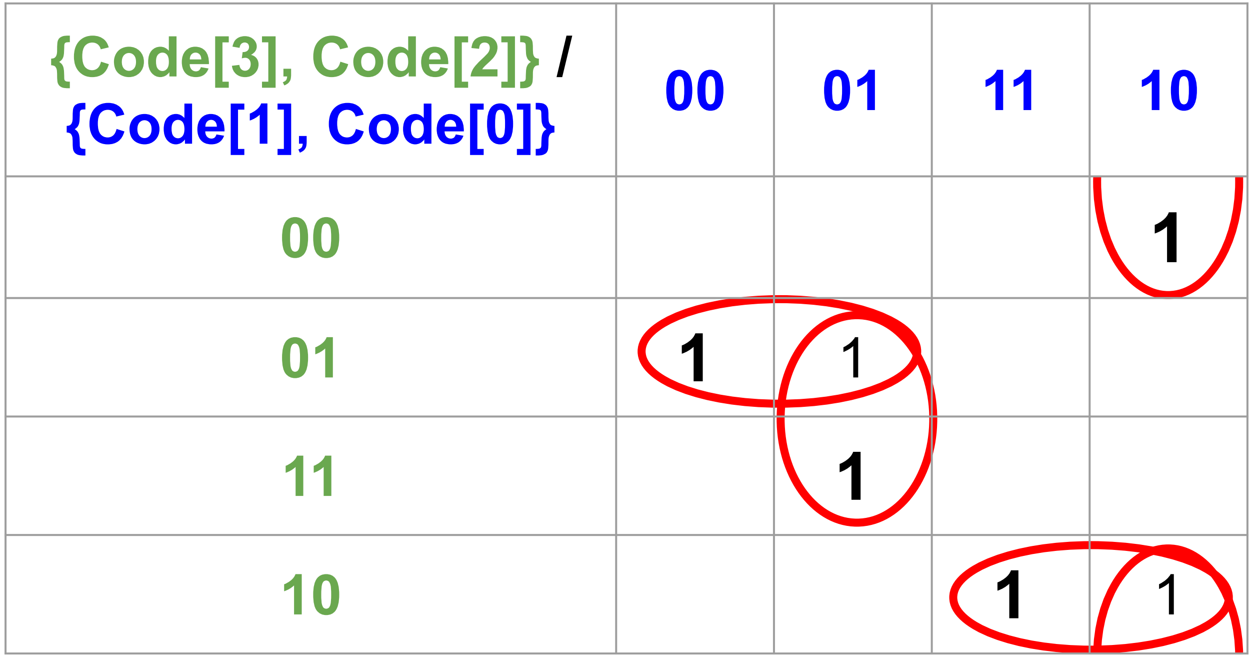
Creating Output Equations (K-Map Approach)

{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

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

Unlocked =

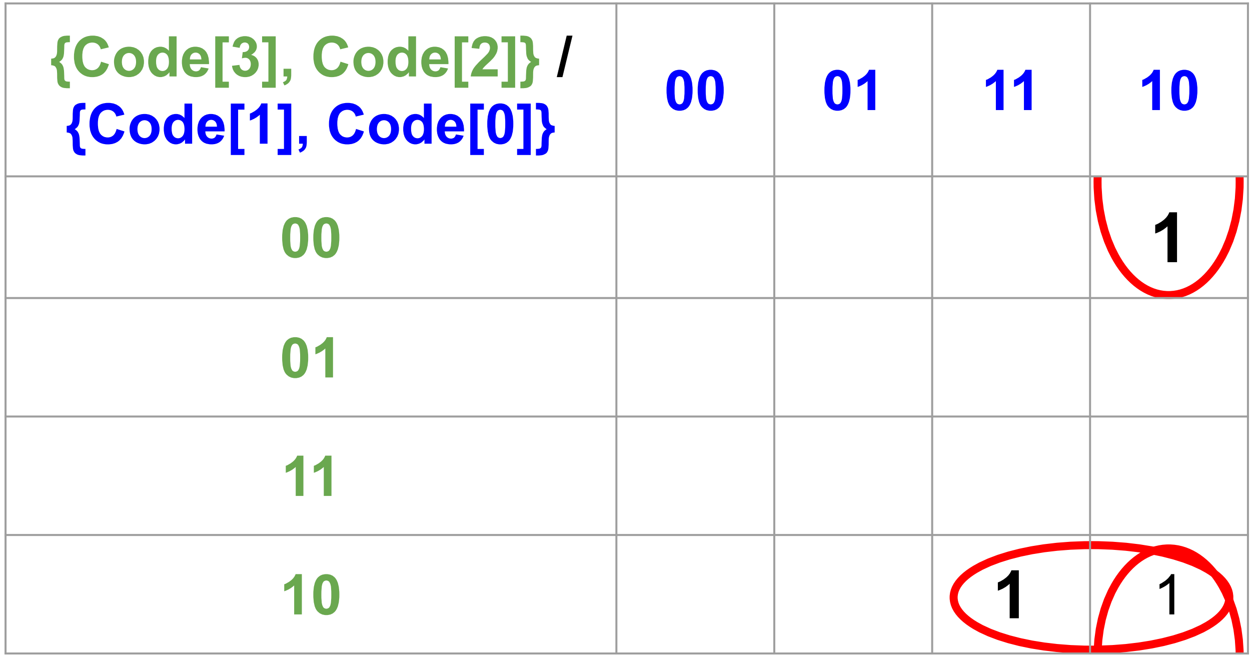
Creating Output Equations (K-Map Approach)

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

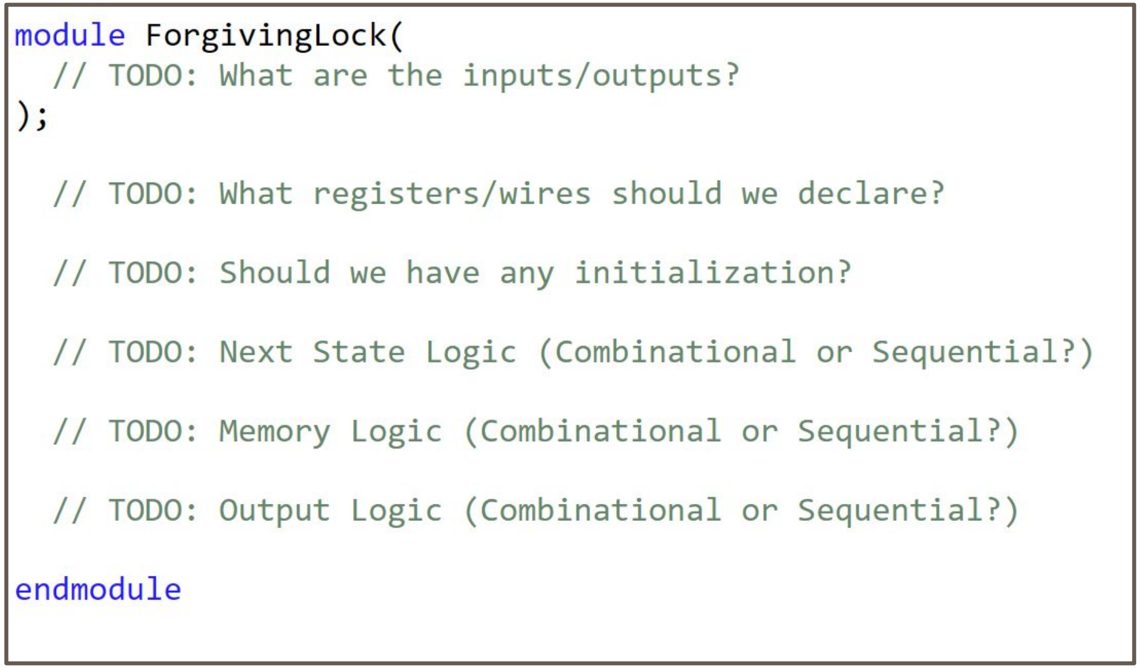
Unlocked = (~Code[3] & Code[2] & ~Code[1]) | (Code[2] & ~Code[1] & Code[0]) | (Code[3] & ~Code[2] & Code[1]) | (~Code[2] & Code[1] & ~Code[0])

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)**



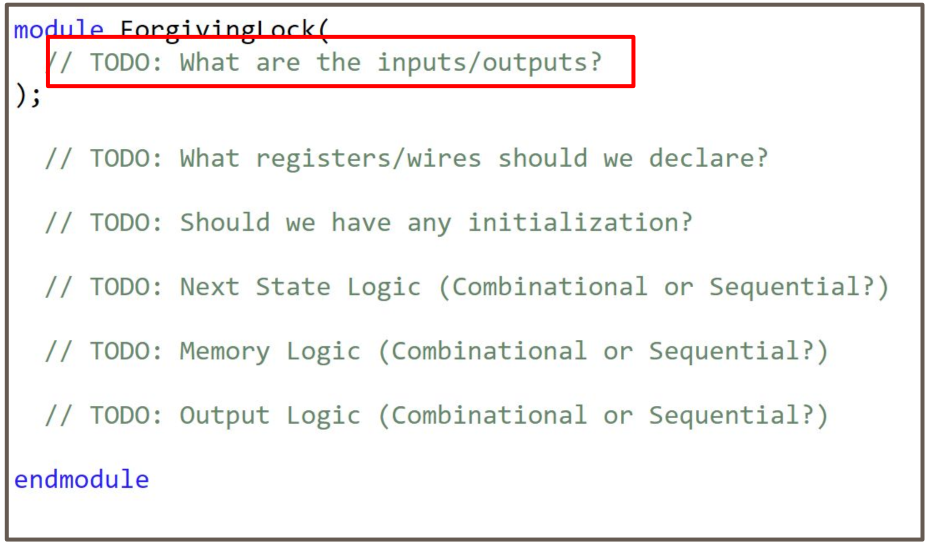
```verilog
module ForgivingLock(
  // TODO: What are the inputs/outputs?
);

  // 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
```

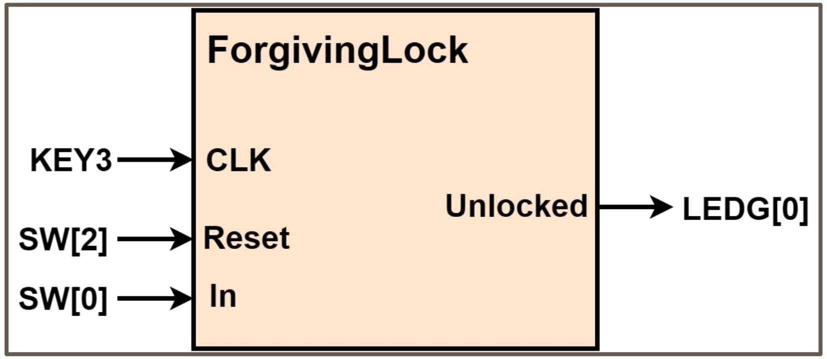
```verilog
module ForgivingLock(
  // TODO: What are the inputs/outputs?
);

  // 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

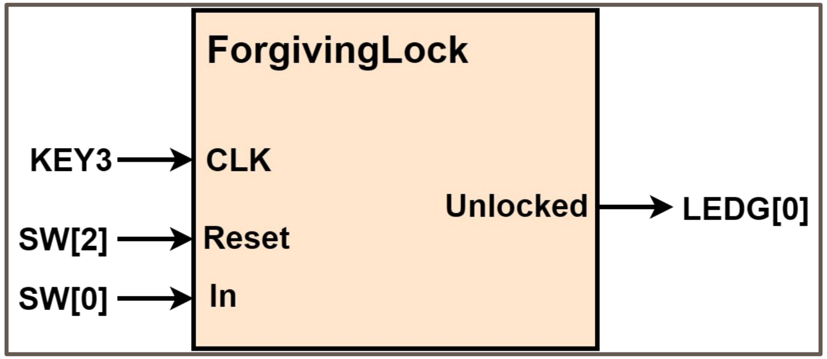
```
module ForgivingTopLevel(
  input [3:3] KEY,
  input [1:0] SW,
  output [0:0] LEDG
);

  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
```

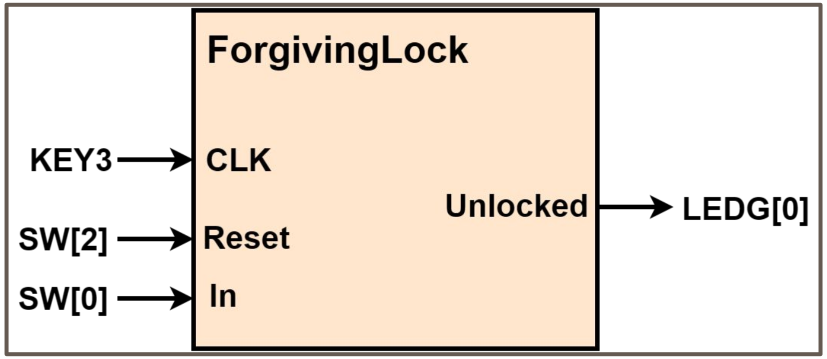

```
module ForgivingLock(
  // TODO: What are the inputs/outputs?
);

  // 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
```

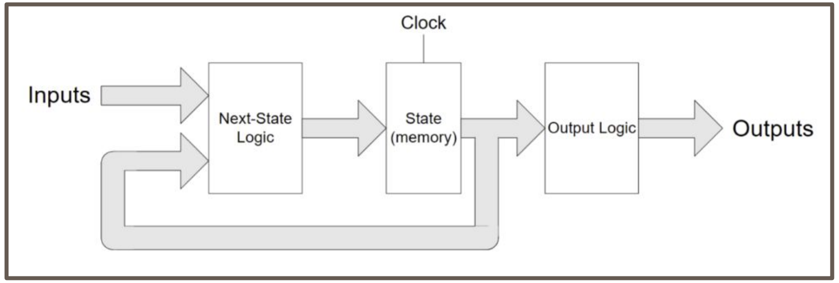


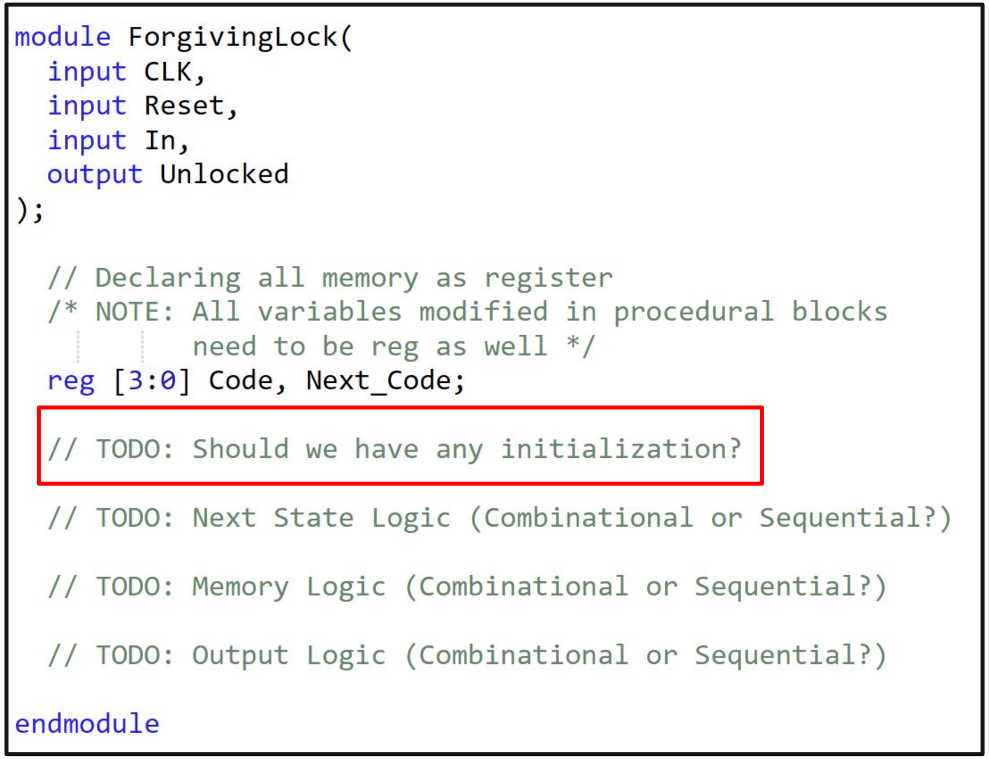
```verilog
module ForgivingLock(
  input CLK,
  input Reset,
  input In,
  output Unlocked
);

  // 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
```



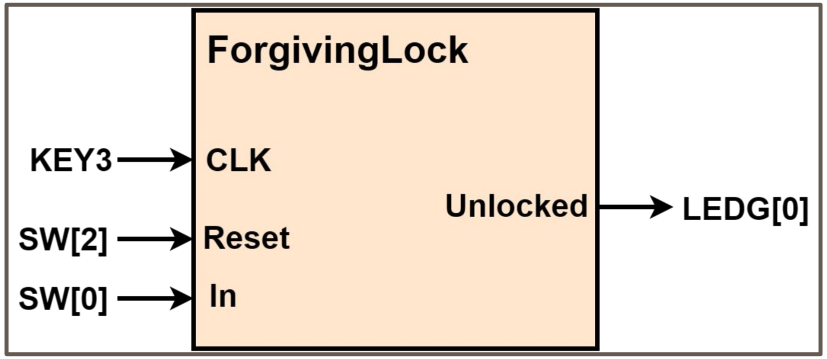
```verilog
module ForgivingLock(
  input CLK,
  input Reset,
  input In,
  output Unlocked
);

  // 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
```

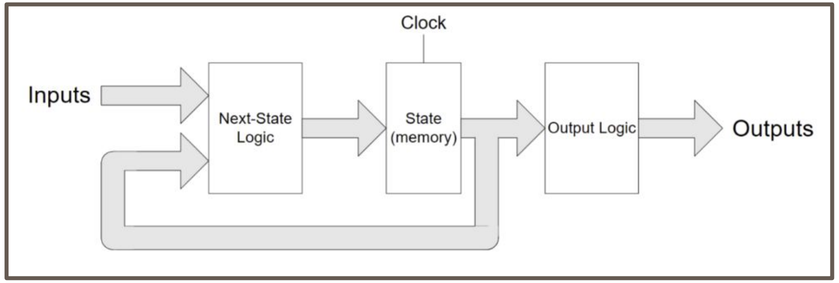




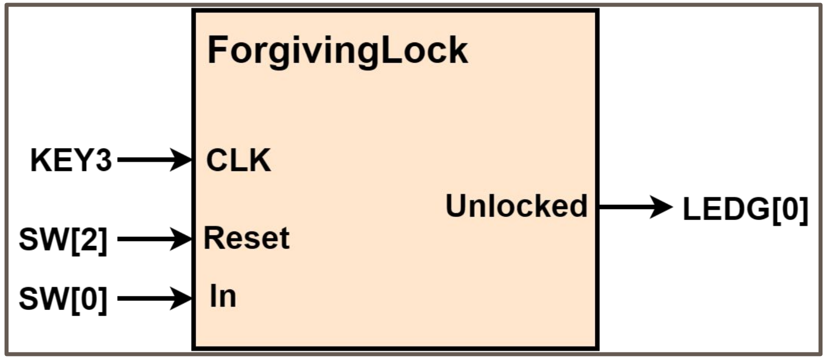


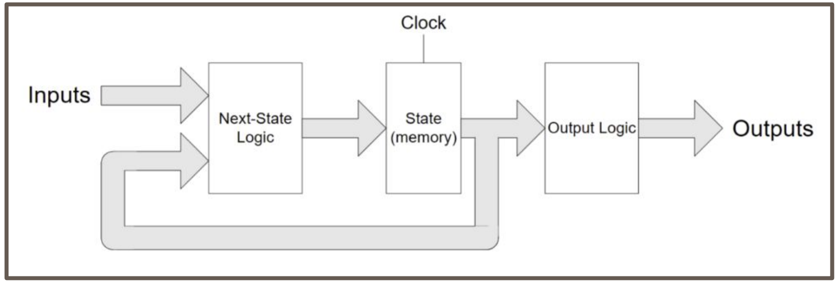
```
module ForgivingLock(
  input CLK,
  input Reset,
  input In,
  output Unlocked
);

  // 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
```

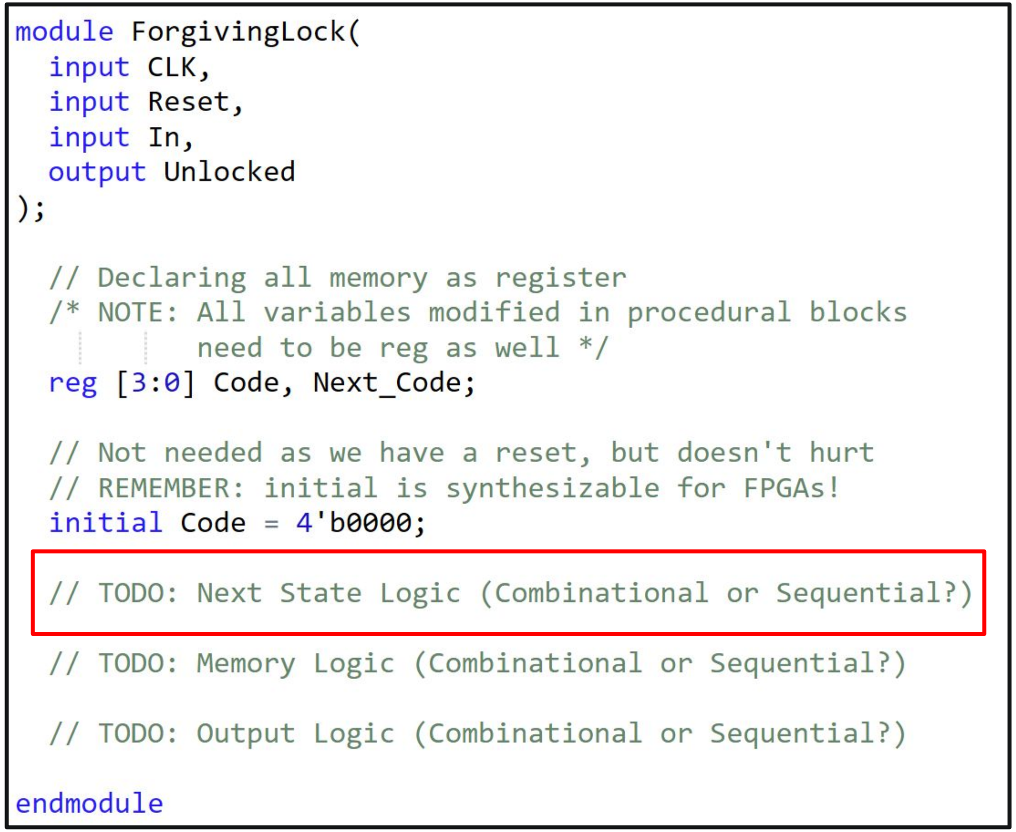




```verilog
module ForgivingLock(
  input CLK,
  input Reset,
  input In,
  output Unlocked
);

  // 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;

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

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

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

endmodule
```







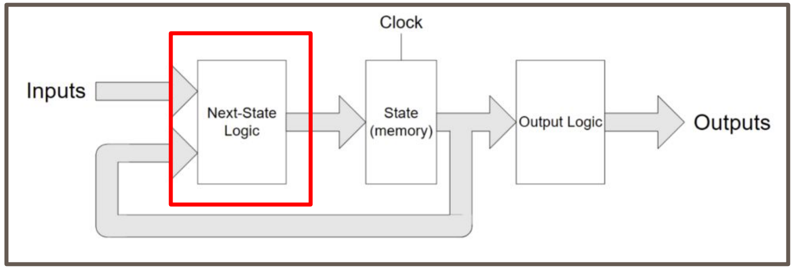
```
module ForgivingLock(
  input CLK,
  input Reset,
  input In,
  output Unlocked
);

  // 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
```

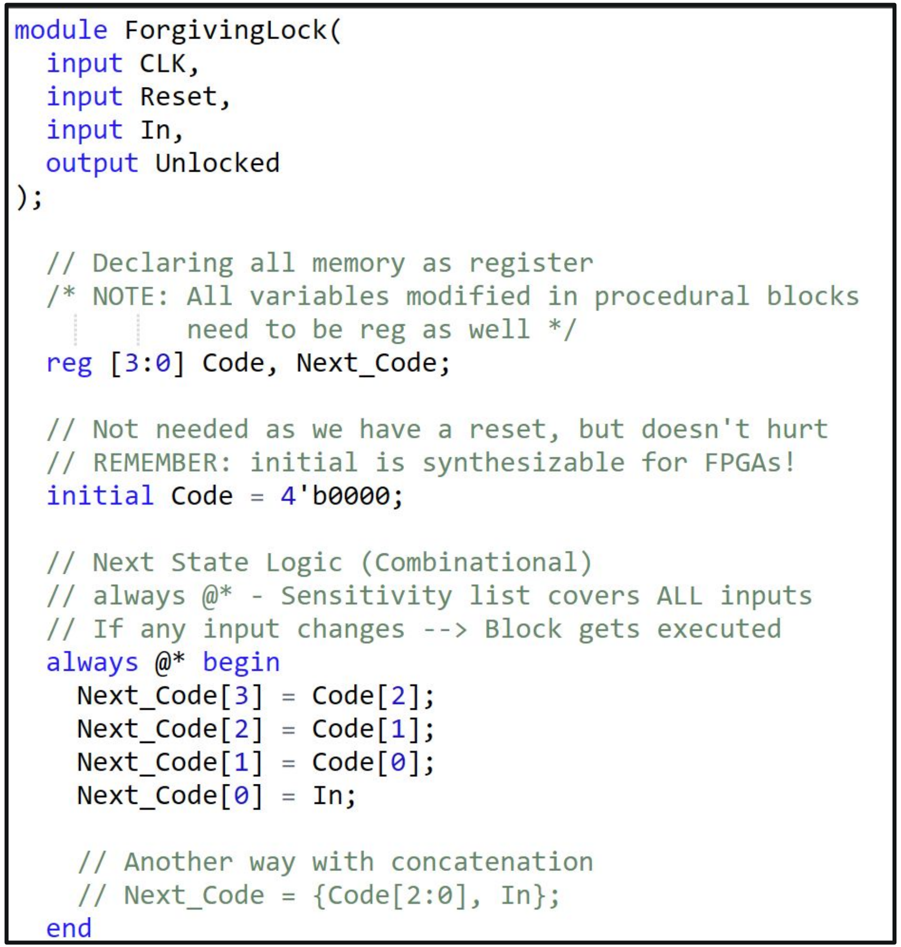
Next_Code[3] = Code[2]

Next_Code[2] = Code[1]

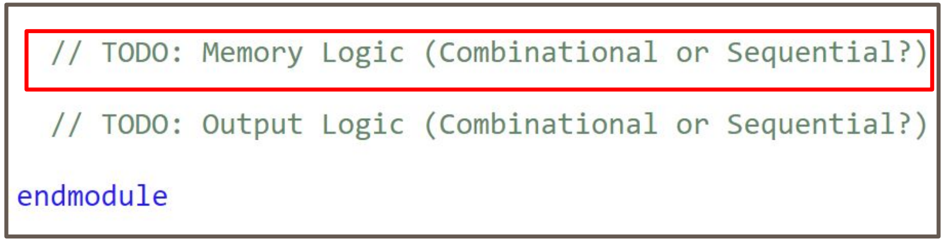
Next_Code[1] = Code[0]

Next_Code[0] = In

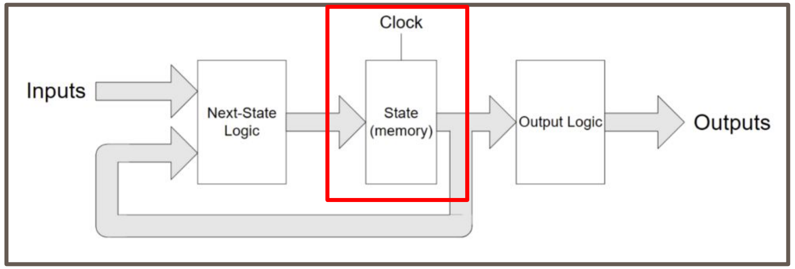


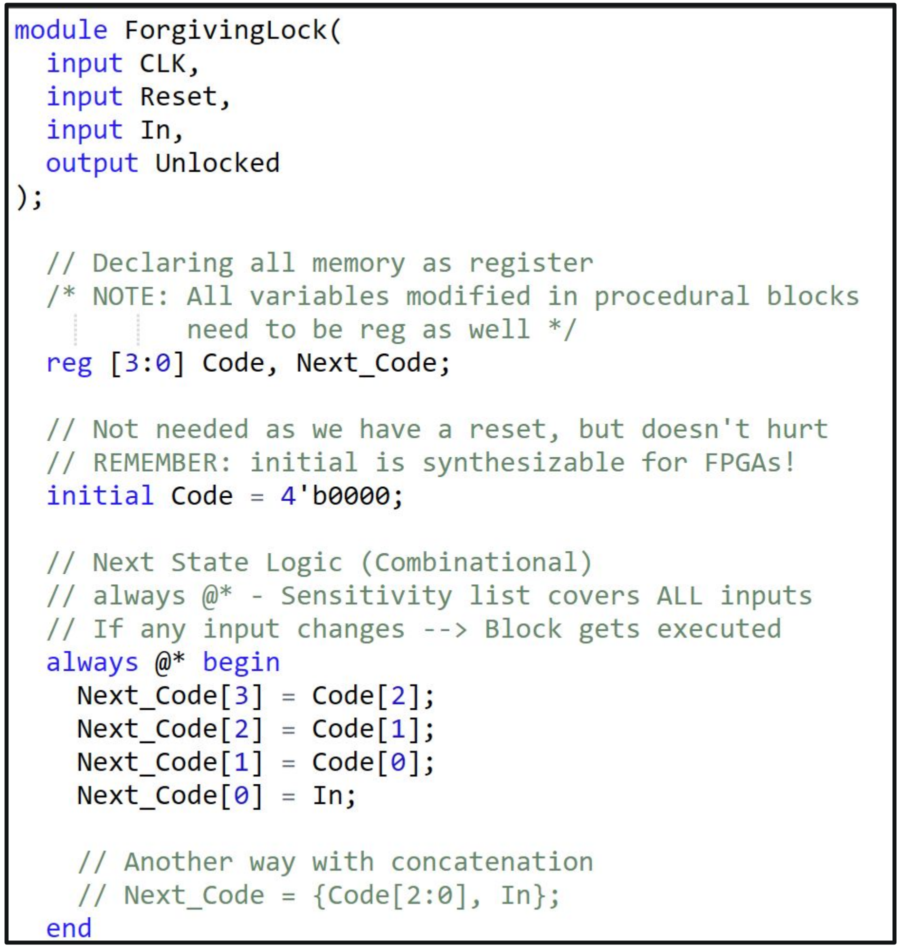


```
module ForgivingLock(
  input CLK,
  input Reset,
  input In,
  output Unlocked
);

  // 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
```

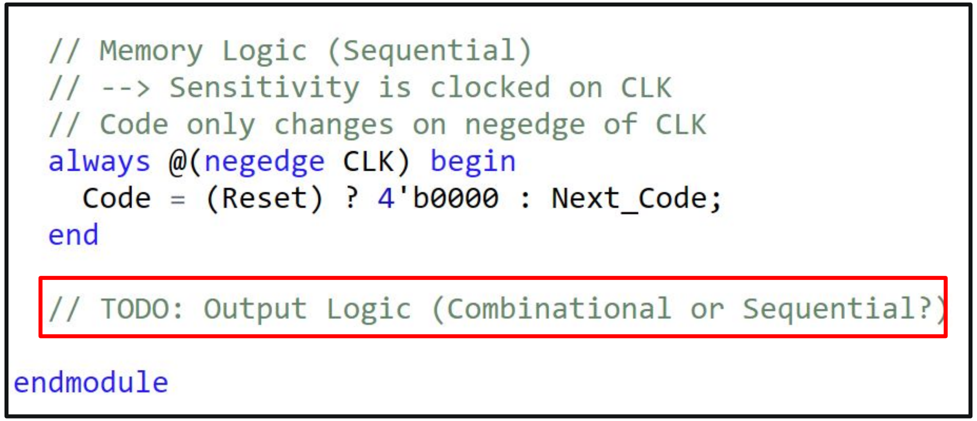


```
  // TODO: Memory Logic (Combinational or Sequential?)

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

endmodule
```

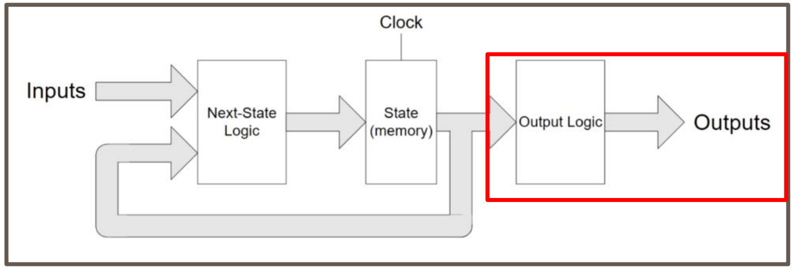




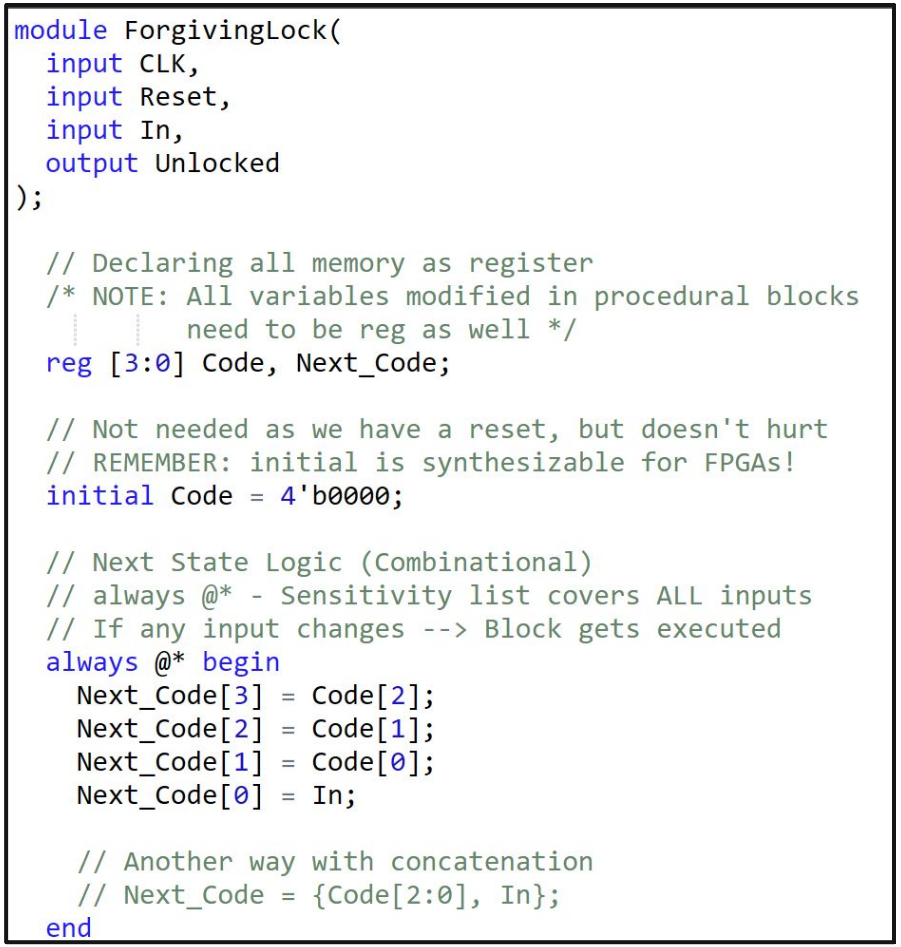
```verilog
module ForgivingLock(
  input CLK,
  input Reset,
  input In,
  output Unlocked
);

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

  // 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
```



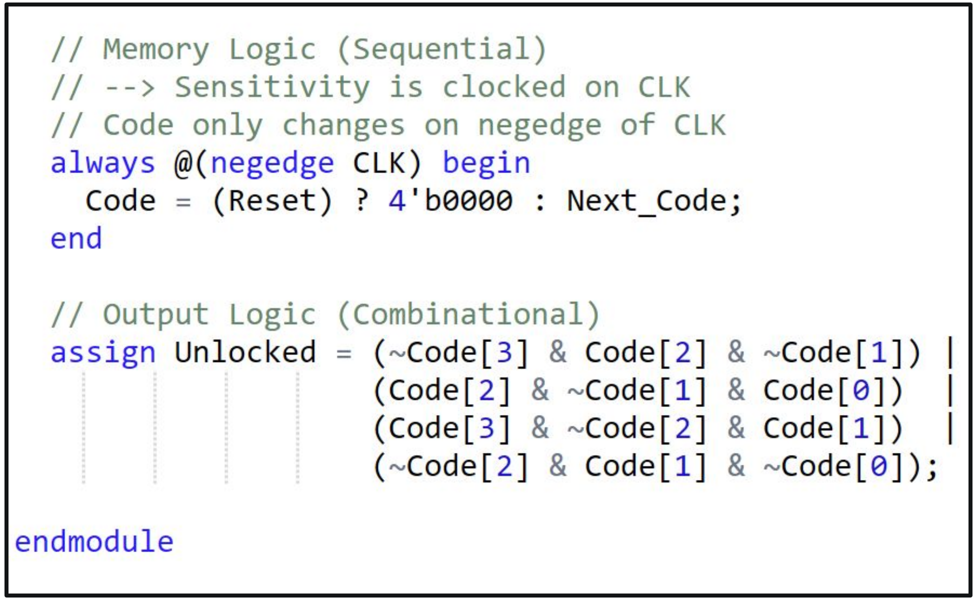
```verilog
  // Memory Logic (Sequential)
  // --> Sensitivity is clocked on CLK
  // Code only changes on negedge of CLK
  always @(negedge CLK) begin
    Code = (Reset) ? 4'b0000 : Next_Code;
  end

  // 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])



```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
  // 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
```