EECS150: Components and Design Techniques for Digital Systems

University of California
Dept. of Electrical Engineering and Computer Sciences

Mid Term 1 – version A — Fall 2004

Last name: ________Solution__________ First name________________________
Student ID: __________________________ Login: _____________________

Lab meeting time: _________________ TA's name: _____________________
(Sorry to ask this next question, but with 100 students packed closely together there may be a wide range of behavior.)
Student to my left is ________________________ ___________________
Student to my right is ________________________ ____________________

No notes. No calculators! This booklet contains 14 numbered pages. Please, no extra stray pieces of paper. The exam contains 6 substantive questions. Browse through it before you start. You have 1.5 hours, so relax, work thoughtfully and give clear answers. Good luck!

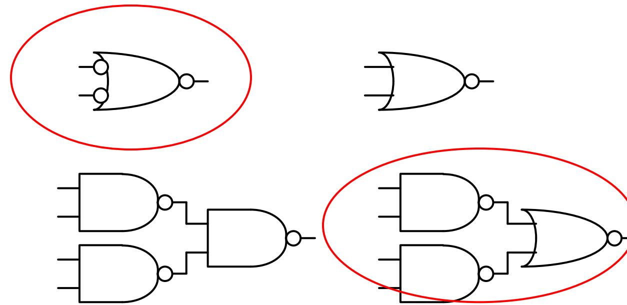
I certify that my answers to this exam are my own work. If I am taking this exam early, I certify that I shall not discuss the exam questions, the exam answers, or the content of the exam with anyone until after the scheduled exam time. If I am taking this exam in scheduled time, I certify that I have not discussed the exam with anyone who took it early.

Signature: ______________________________________

Problem	Score
Problem 1 [15]	
Problem 2 [10]	
Problem 3 [10]	
Problem 4 [15]	
Problem 5 [20]	
Problem 6 [30]	
Total [100]	

Problem 1. [15]

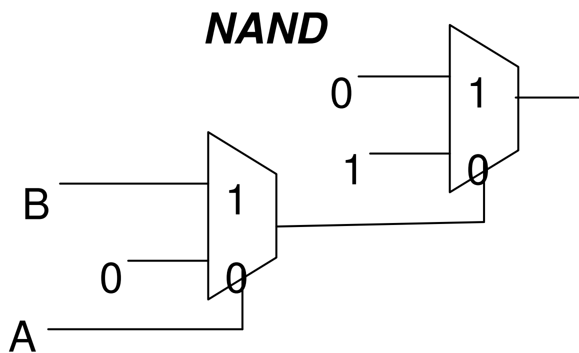
1.a. Circle the gate-level circuits below that implement a Boolean AND function?



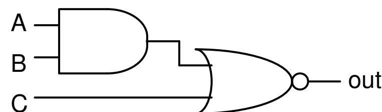
Circuits are in another order for B test

1.b. Show that a 2-to-1 MUX is universal, i.e. that any Boolean expression can be implemented with a collection of 2-to-1 multiplexers.

You could implement AND, OR, NOT, or just NAND or NOR



1.c. Write a Verilog module that implements the following combinational logic.




```
module (A, B, C, out);
      input        A, B, C;
      output       out;

      wire x;

      and (x, A, B);
      nor  (out, x, C);

endmodule
```

```
module (A, B, C, out);
 input       A, B, C;
 output      out;

 assign C = ~(A & B)|C);

endmodule
```

```
module (A, B, C, out);
   input A, B, C;
   output     out;

   reg out;

   always @(A or B or C)
      out = ~(A & B)|C);

endmodule
```

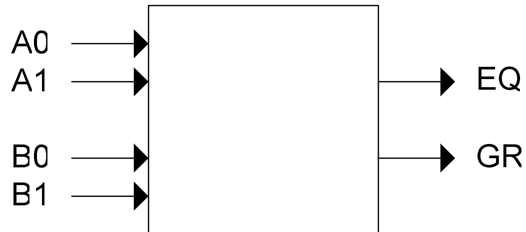
1.d. Write a Verilog module that implements a D flip-flop with reset.

```
module (D, Q, Clock, Reset);
      input       D, Clock, Reset;
      output      Q;

      reg Q;

      always @ (posedge Clock) begin
         if (Reset) Q <=      1`b0;
         else Q <=            D;
      end



endmodule
```

Problem 2. [10]

Generate a truth table with appropriate don't-cares for the circuit shown below. It has two 2-bit unsigned inputs A = {A_1,A_0} and B = {B_1,B_0} and two outputs, EQ, and GR.

EQ is 1 if A = = B and 0 otherwise. GR is 1 when A > B and GR is 0 when A < B. Your solution should be concise and it should allow for an efficient implementation.



A1	A0	B1	B0	EQ	GR
0	0	0	0	1	x
		0	1	0	0
		1	x	0	0
0	1	0	0	0	1
		0	1	1	x
		1	x	0	0
1	0	0	x	0	1
		1	0	1	x
		1	1	0	0
1	1	0	x	0	1
		1	0	0	1
		1	1	1	x

GR is inverted for version B

Problem 3. [10]

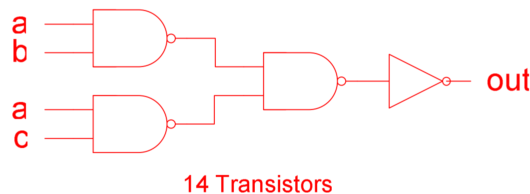
You are to implement the following Boolean operation over three inputs:

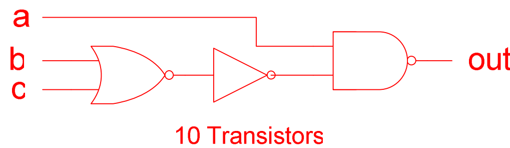
$out = (\overline{ab}) \bullet (\overline{ac})$

'a' and 'b' are swapped on B test.

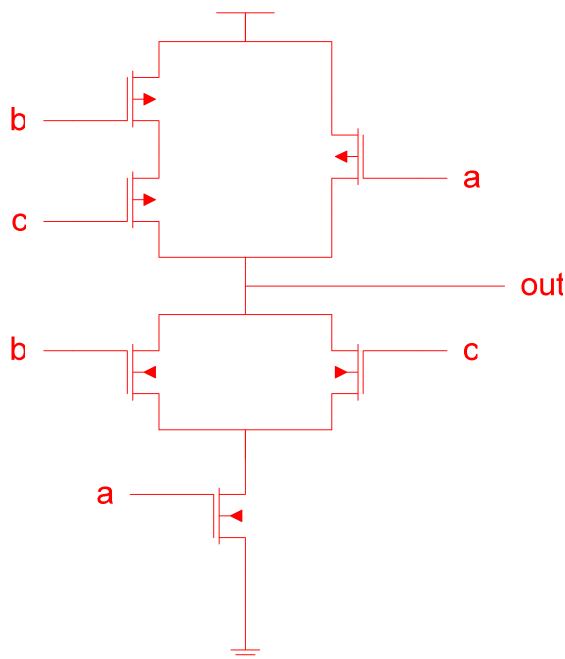
3.a Implement this efficiently using NAND, NOR, and INV gates.
See below…

3.b. How many n-channel and p-channel transistors are used in this gate-level implementation?





3.c. How much can you reduce this number by implementing the operation directly at the transistor level? Draw a transistor-level schematic.



Question 4: [15] The following is a truth table for a 4-input, 2-output logic function:
Inputs: **a**, **b**, **c**, **d**, Outputs: **x**, **y**

a	**b**	**c**	**d**	**x**	**y**
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	-	1
0	0	1	1	1	0
0	1	0	0	1	1
0	1	0	1	1	1
0	1	1	0	0	1
0	1	1	1	0	-
1	0	0	0	-	0
1	0	0	1	1	1
1	0	1	0	-	1
1	0	1	1	0	0
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	1	-
1	1	1	1	1	0

4.a. Compute minimal sum-of-product (SOP) expressions for **x** and **y using kmaps**.

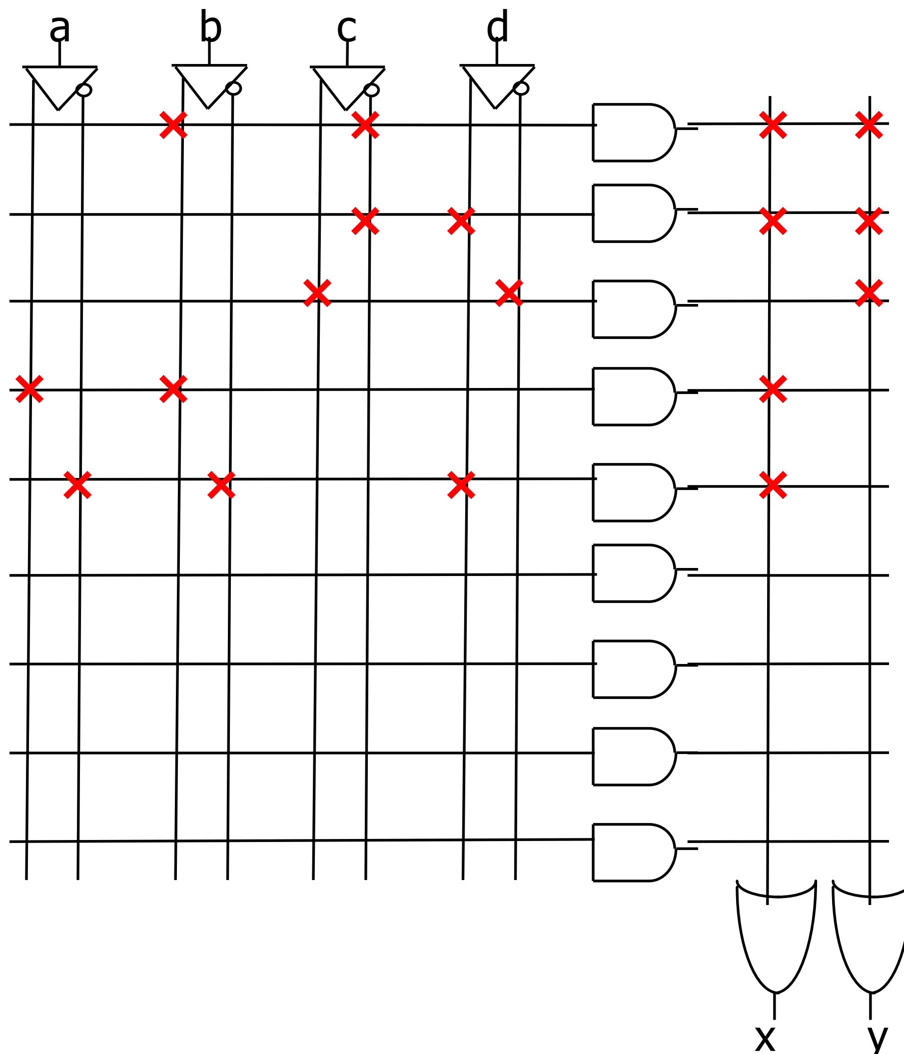
x	cd	~c~d	~cd	cd	c~d
ab		00	01	11	10
~a~b	00	0	1	1	-
~ab	01	1	1	0	0
ab	11	1	1	1	1
a~b	10	-	1	0	-

$$x = ab + b\bar{c} + \bar{c}d + \bar{a}\bar{b}d$$

y	cd	~c~d	~cd	cd	c~d
ab		00	01	11	10
~a~b	00	0	1	0	1
~ab	01	1	1	-	1
ab	11	1	1	0	-
a~b	10	0	1	0	1

$$y = c\bar{d} + b\bar{c} + \bar{c}d$$

4.b. In this part, you will implement your logic on a PLA On the diagram below. You want to **minimize** the number of AND gates (rows) on the PLA. Mark the utilized connections on the PLA diagram for computing **x** and **y**.
(*Note: you may not find use for all the rows or cols on the PLA.*)



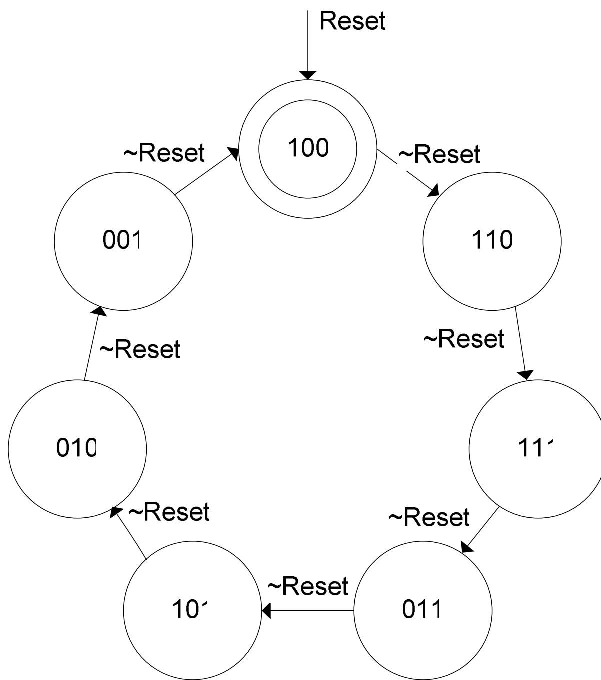
The key here was to share minterms b~c and ~cd

Problem 5 [20].

You are to implement a 3 bit counter with the following (somewhat unusual) state transition diagram.

	$C_2C_1C_0$	
After reset:	1 0 0	
After 1 Clock Cycle:	1 1 0	
After 2 Clock Cycle:	1 1 1	
After 3 Clock Cycle:	0 1 1	
After 4 Clock Cycle:	1 0 1	
After 5 Clock Cycle:	0 1 0	
After 6 Clock Cycle:	0 0 1	
After 7 Clock Cycle:	1 0 0	<-- Sequence repeats
After 8 Clock Cycle:	1 1 0	

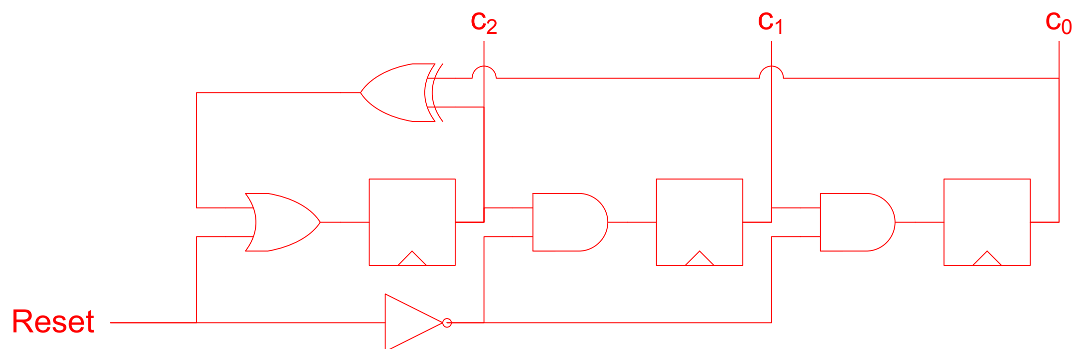
.
.
.



5.a. Draw the schematic diagram for this 3-bit counter. You may only use 1-bit flip flops and primitive gates. Make sure to implement the "Reset" input, as your flip-flops do not have a built in reset input.

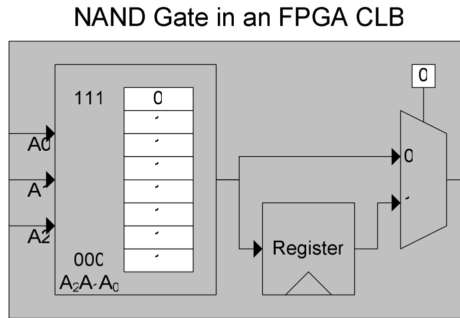
CS	**NS**
$C_2C_1C_0$	**$N_2N_1N_0$**
000	XXX
001	100
010	001
011	101
100	110
101	010
110	111
111	011

$N_2 = C_0$ xor C_2
$N_1 = C_2$
$N_0 = C_1$



5.b. Shown below is an example CLB from an FPGA filled in to implement a NAND gate. Notice that not only is the 3-LUT filled in, but the control bit for the MUX is set.

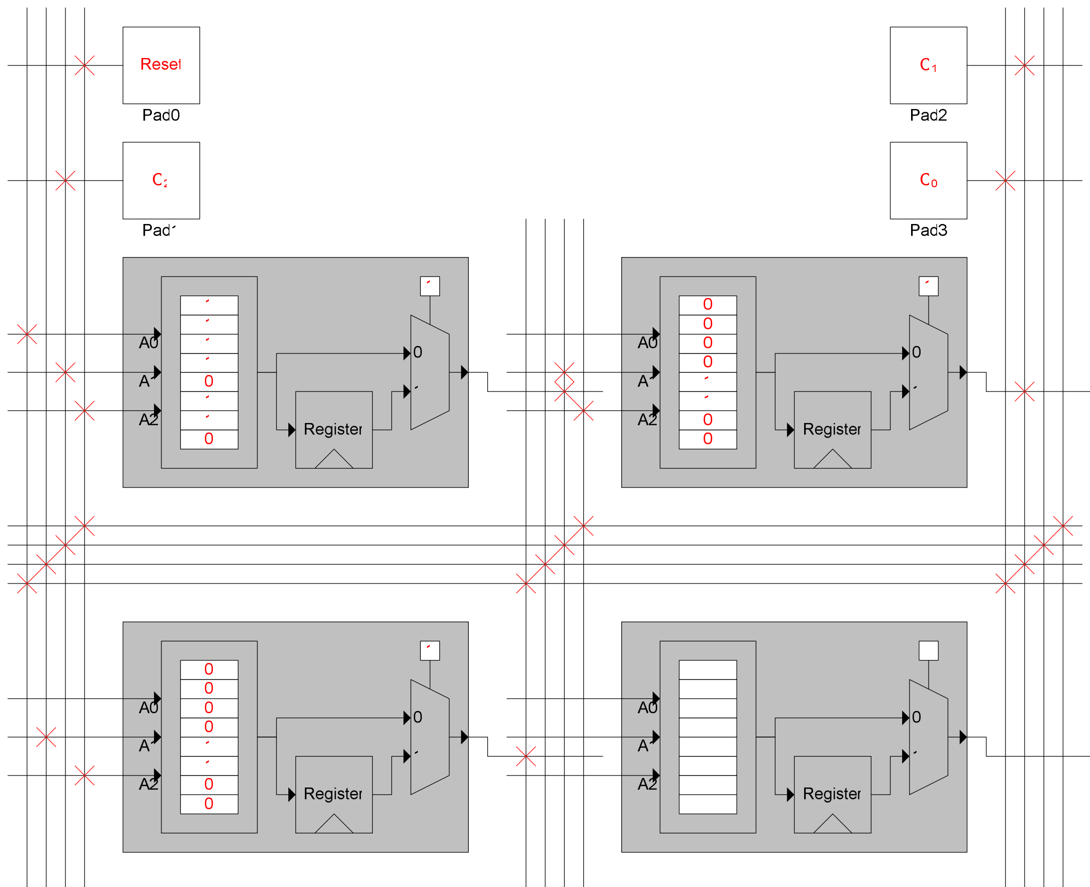
NAND Gate in an FPGA CLB



In this problem you must implement your counter in the simplified FPGA below. Fill in the white boxes with either 1 or 0 to indicate both the programming of the 3-LUTs and the mux control bits. Indicate connected wires with an X as with PLAs.

In addition to configuring the CLBs you must make sure to route all the signals you use, including "Reset" and to configure the four I/O pads at the top.

Each signal which must connect to the outside world must be connected to an I/O pad. "Reset" is the only input and three bits of the counter "$C_2C_1C_0$" are the only outputs. In the white I/O pad box, write in the name of the signal connected to it. Each I/O pad would be connected to a pin on an FPGA chip.



Problem 6 [30]. In this problem you will be working with a new combination lock. You will design the controller for a lock with 11 buttons labeled ‘0’-‘9’ and “Reset”. To open the lock, the user must press the correct numbered buttons in sequence. The “Reset” button should return the lock to a default state at any time.

The lock must respond with either the “Open” or “Locked” output at all times. The lock should not output “Open” until one of the combinations has been entered. Once the lock is opened and outputting “Open” it should continue to do so until “Reset.”

If at any time the user enters even a single wrong digit the lock should output “Error” and continue to do so until it is “Reset.”. Whenever the lock is not open it should assert the “Locked” output. The lock may report both “Error” and “Locked” but never “Open” and “Locked” or “Open” and “Error.”

You may assume that the inputs from the buttons are high for only one cycle when the button is pushed regardless of the clock speed and how long the button was held down.

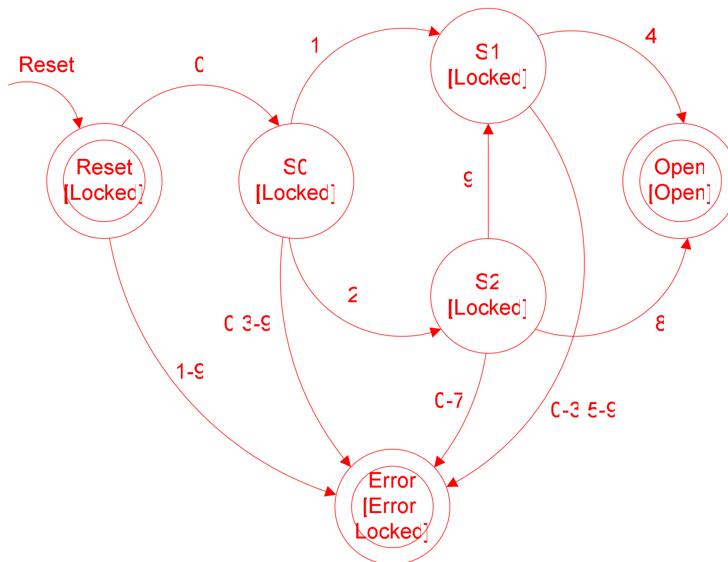
As a twist, the lock must respond with “Open” to any of three different combinations:

- 0 – 1 – 3
- 0 – 6 – 5 – 3
- 0 – 6 – 9

Entering any one of the above three combinations should cause the lock to “Open.”

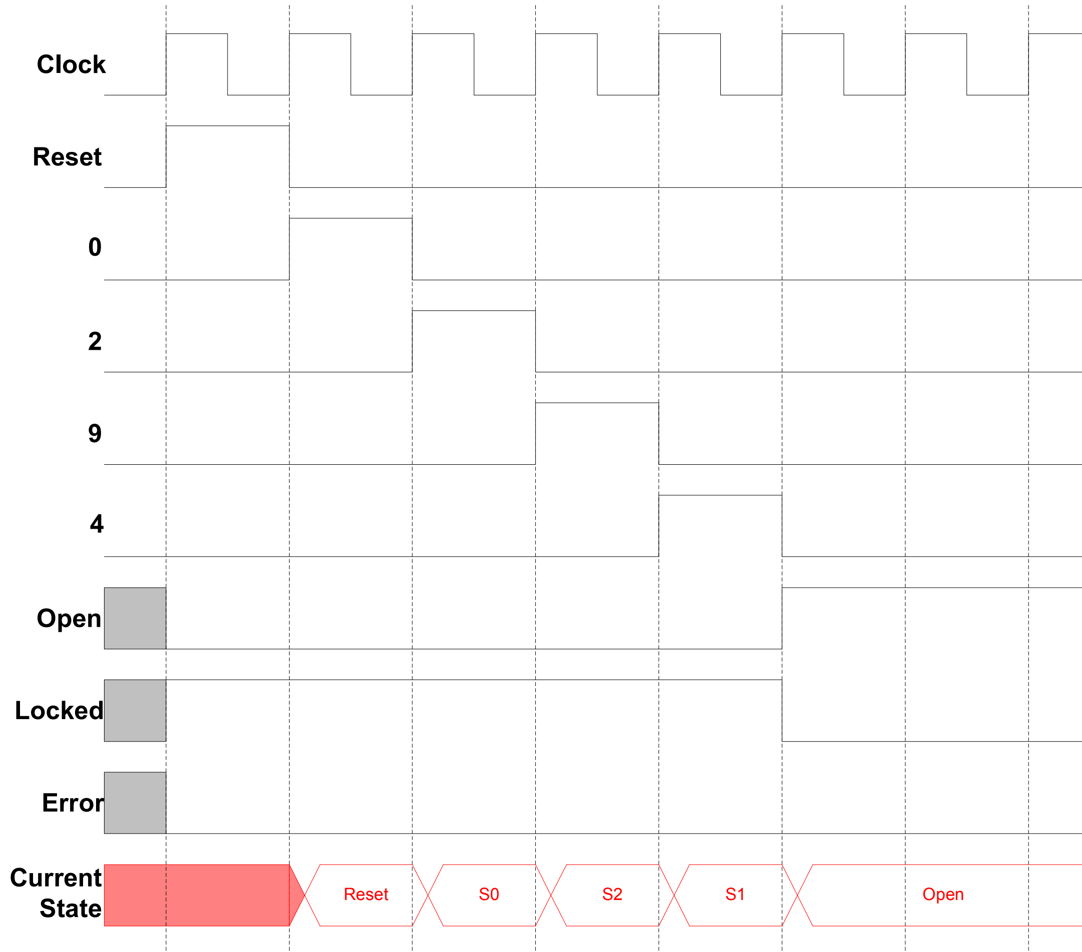
6.a. Draw the bubble and arc diagram for your **Moore machine** implementation of the controller.

I. Label and name all states appropriately
II. Label all arcs with the buttons that will cause that transition
   a. You may label arcs with ranges of buttons to save time
III. Label all states with the outputs that should be asserted in that state
   a. Outputs are assumed to be unasserted unless otherwise marked



6.b. Fill in the current state values for each cycle in the below timing diagram. You should use the state names from your bubble-and-arc diagram in part (a).

Note that the gray boxes in the following diagram stand for "Don't Know/Don't Care"



6.c. Fill in the verilog shell given below with an implementation of your controller.

Note that we have added a signal called "Input" for your use. It is the OR of all of the 10 number inputs. That is to say, it will be 1'b1 when ANY of the number buttons are pressed.

```
module LockFSM(Numbers, Reset, Clock, Open, Locked, Error);
	input [9:0]	Numbers;
	input		Reset;
	input		Clock;

	output		Open, Locked, Error;

	wire		Input;

	assign	Input =		(|Numbers);

	localparam	STATE_Reset =	3'b000,
			STATE_S0 =	3'b001,
			STATE_S1 =	3'b010,
			STATE_S2 =	3'b011,
			STATE_Open =	3'b100,
			STATE_Error =	3'b101,
			STATE_X =	3'bxxx;

	reg	[2:0]	CurrentState, NextState;

	assign	Open =		(CurrentState == STATE_Open);
	assign	Locked =	(CurrentState != STATE_Open);
	assign	Error =		(CurrentState == STATE_Error);

	always @ (posedge Clock) begin
		if (Reset) CurrentState <=		STATE_Reset;
		else if (Input) CurrentState <=		NextState;
	end

	always @ (*) begin
		NextState =				CurrentState;

		case (CurrentState)
			STATE_Reset: begin
				if (Numbers[0]) NextState =	STATE_S0;
				else NextState =		STATE_Error
			end
			STATE_S0: begin
				if (Numbers[1]) NextState =	STATE_S1;
				else if (Numbers[2]) NextState = STATE_S2;
				else NextState =		STATE_Error;
			end
			STATE_S1: begin
				if (Numbers[4]) NextState =	STATE_Open;
				else NextState =		STATE_Error;
			end
			STATE_S2: begin
				if (Numbers[8]) NextState =	STATE_Open;
				else (Numbers[9]) NextState = STATE_S1;
				else NextState =		STATE_Error;
```

```
            end
            STATE_Open: NextState =             STATE_Open;
            STATE_Error: NextState =            STATE_Error;
            default: NextState =                STATE_X;
        endcase
    end

endmodule
```