University of New South Wales

School of Electrical Engineering and Telecommunications

INDIVIDUAL Assessment Task: ass2

Course Code	ELEC2141	Course Name	Digital Circuit Design
Week/Session/Year	2024 T1	Lecturer	Beena Ahmed

Student Number	5480710
Family Name	Luo
Given Names	Chengji

Mark/Grade given (For official use only)

Marker's Comments:

Since this work counts toward your formal assessment for this course, please write your name and student number where indicated above, and sign the declaration below. Attach this cover sheet to the front of your submission, so that your name and student number can be seen without any cover needing to be opened.

For further information, please see http://www.lc.unsw.edu.au/plagiarism/index.html

---

I declare that this assessment item is my own work, except where acknowledged, and has not been submitted for academic credit elsewhere, and acknowledge that the assessor of this item may, for the purpose of assessing this item:

- ***Reproduce this assessment item and provide a copy to another member of the University ; and/or,***
- ***Communicate a copy of this assessment item to a plagiarism checking service (which may then retain a copy of the assessment item on its database for the purpose of future plagiarism checking).***

I certify that I have read and understood the University Rules in respect of Student Academic Misconduct.

Signature of student _____ Date: 22nd Apr 2024

ELEC2141 Assignment

2024 T1

1 Introduction

In this assignment we are tasked to build a functional keypad in order to secure Mr Keggs' brewery at night. This will be done by drawing a state diagram and coding the logic in Verilog, as well as including simulation data to make sure it can be implemented efficiently without any errors.

2 Identify the system inputs and outputs

The necessary inputs and outputs for the implementation can be seen below:

- **Inputs:**
  - clk - the clock function that will provide the enable for the flip-flop trigger.
  - reset - our reset signal that clears all the memory in our flip flops.
  - keypad[11:0] - a 12 bit keypad that is used to operate the locking mechanism.

- **Outputs:**
  - lock - a single bit output that determines whether our lock is active.
  - alarm - a single bit output that determines whether the alarm is on.

3 Draw a state diagram for the FSM

I implemented the logic of this circuit with multiple finite state machines to separate the logic and make the circuit more simple. The following state diagrams that make up the hierarchy can be seen below.

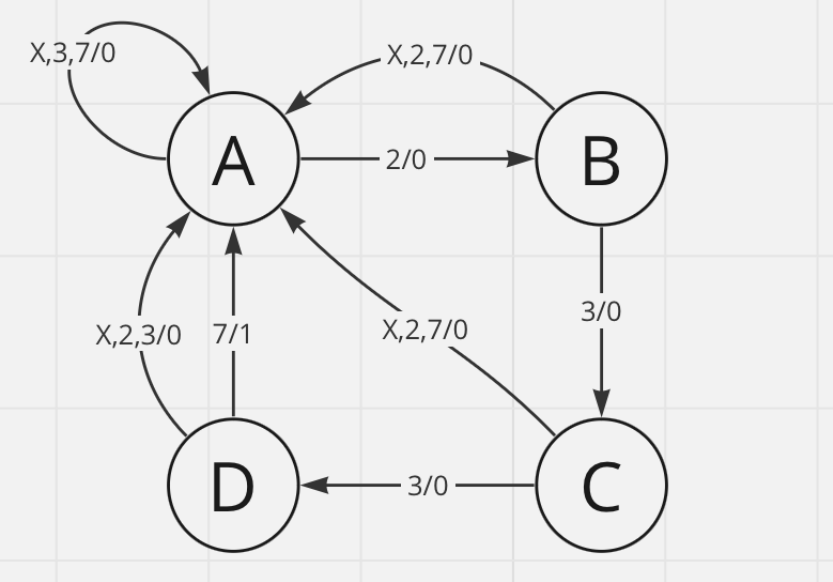


Figure 1: keypad inputs

The above state diagram describes the logic that is used to determine the correct sequence of inputs from the keypad. We will have 4 inputs - 2, 3, 7 and X, where X corresponds to all the keypad numbers that will provide a wrong entry in the passcode (0,1,4,5,6,8,9). There is also a 1-bit output 'correct' which is HIGH when the correct 4 digit passcode is entered. This logic can be implemented with four states:

- **A - 00** This state is the 'idle' state where we initially start. An input of 2 would bring us to state B whereas any other input would remain in state A. It has an output of 0 because the correct passcode has not been entered.
- **B - 01** This state is a secondary state. An input of 3 would bring us to state C whereas any other input would reset it to the idle state. Output = 0.
- **C - 10** This is the third state and also detects the third digit in the passcode. An input of 3 would bring us to state D whereas any other input would reset it to the idle state. Output = 0.
- **D - 11** This is the state that detects the fourth digit of the passcode. The next state is always A, however if you input a 7, the 'correct' output would be HIGH.

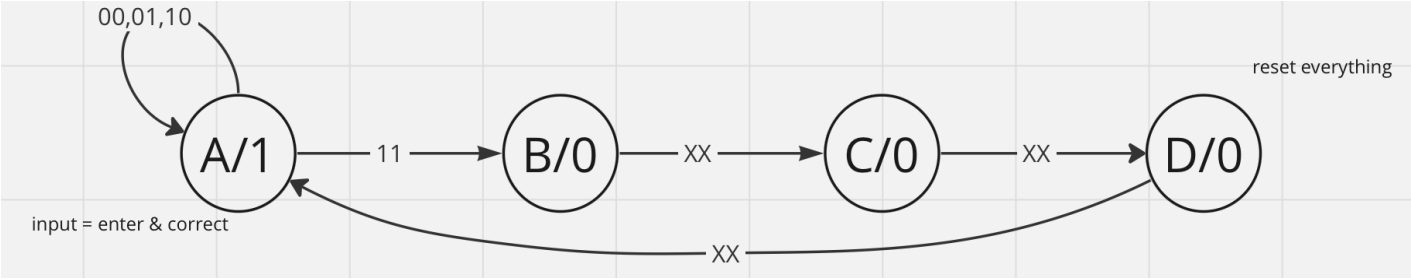


Figure 2: lock timer

The lock timer state diagram is a simple FSM that counts 3 clock cycles after a HIGH 'correct' and 'enter' input has been received. This would then **output** 'lock' = LOW for state B, C, and D. After it reaches the 4th state D, it jumps back to the first state and re-locks.

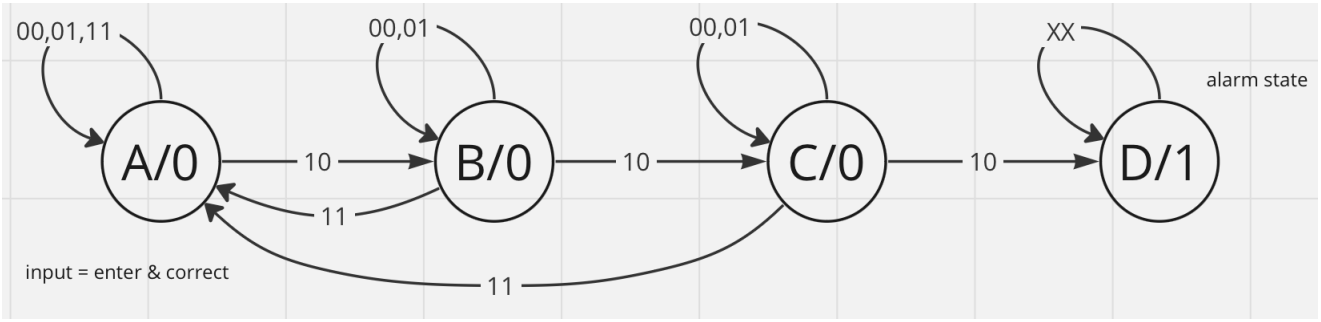


Figure 3: error counter

The error counter is a Moore simple state diagram that implements the 'error count' logic where the alarm will turn on after 3 incorrect attempts. It achieves this using 4 states and a 2 bit input X_1X_0 where X_1 = enter and X_0 = correct. There is also a 1-bit output 'alarm'. The state logic can be seen as following:

- **A - 00** This state is our initial 'idle' state. If our input is incorrect and an 'enter' is pressed the error count will increase by 1 and our next state becomes B. In this case, our alarm output would be LOW.
- **B - 01** State B also has an alarm output of LOW. If another incorrect attempt is entered, the next state would be state C. However, if a correct input is detected, we return to state A.
- **C - 10** Same logic as state B. Here the error count is 2 and the alarm is still LOW. If a third incorrect attempt is entered in a row an alarm will turn on (state D).
- **D - 11** This is the alarm state. Note how we will stay in this state regardless of our inputs. This will be perpetual until a reset signal is applied (either manually or from getting the passcode correct).

4 State tables / Optimisation

4.1 Keypad Inputs

In order to effectively optimise the implementation above, I have created several state tables to describe the flow of logic. These can further be used to create implication tables to determine if the states can be reduced.

State Table						
Present state		**Inputs**		**Next state**		**Outputs**
Q_1	Q_0	X_1	X_0	$Q_1(t+1)$	$Q_0(t+1)$	Correct
A ~ 00		0	0	0	1	0
		0	1	0	0	0
		1	0	0	0	0
		1	1	0	0	0
B ~ 01		0	0	0	0	0
		0	1	1	0	0
		1	0	0	0	0
		1	1	0	0	0
C ~ 10		0	0	0	0	0
		0	1	1	1	0
		1	0	0	0	0
		1	1	0	0	0
D ~ 11		0	0	0	0	0
		0	1	0	0	0
		1	0	0	0	1
		1	1	0	0	0

The above state table describes the state transitions of the state diagram in *figure 1* **keypad inputs**. I mapped the four states with a 2-bit binary code. This compiles all the logic used above and presents the next state and output, allowing us to go down into a structural level and create K-maps for flip flop implementation. An implication table has been included below to show that the states cannot be reduced.

Implication table			
B	X		
C	X	X	
D	X	X	X
	A	B	C

Moving forward, the state tables for the JK flip-flop, D flip-flop, and T flip-flop can be derived from the above state table through a careful consideration of present and next states. The **Boolean logic** for all the inputs can then be calculated using K-maps.

4.1.1 JK flip-flop implementation

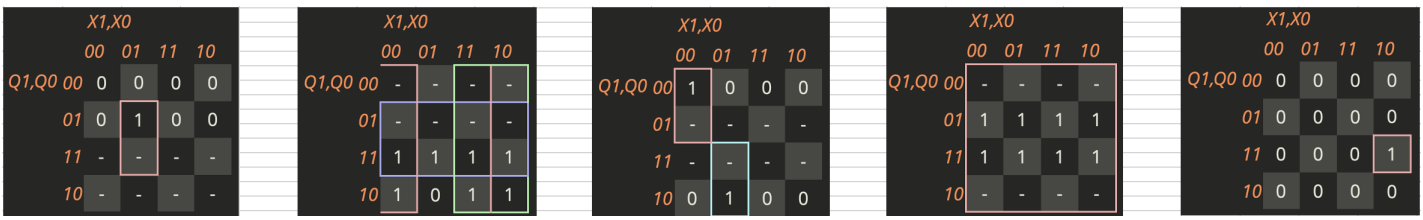
State Table - JK Flip-flop							
Present state		**Inputs**		**Next state**		**Outputs**	**J & K**	
Q_1	Q_0	X_1	X_0	$Q_1(t+1)$	$Q_0(t+1)$	Correct	J_1K_1	J_0K_0
A ~ 00		0	0	0	1	0	0X	1X
		0	1	0	0	0	0X	0X
		1	0	0	0	0	0X	0X
		1	1	0	0	0	0X	0X
B ~ 01		0	0	0	0	0	0X	X1
		0	1	1	0	0	1X	X1
		1	0	0	0	0	0X	X1
		1	1	0	0	0	0X	X1
C ~ 10		0	0	0	0	0	X1	0X
		0	1	1	1	0	X0	1X
		1	0	0	0	0	X1	0X
		1	1	0	0	0	X1	0X
D ~ 11		0	0	0	0	0	X1	X1
		0	1	0	0	0	X1	X1
		1	0	0	0	1	X1	X1
		1	1	0	0	0	X1	X1

This table describes the J and K inputs that are required to implement the state diagram in *figure 1* where I used two JK flip-flops for the four states mapped to a 2-bit binary code. Thus the J and K inputs are derived from the relation between the **present state** and the **next state**. This logic is based on the excitation table for a single JK flip-flop defined below.

Q_n	Q_{n+1}	J	K
0	0	0	X
1	0	X	1
0	1	1	X
1	1	X	0

$$Q(n+1) = J\overline{Q} + \overline{K}Q$$

Subsequently, the full circuit can be calculated using the K-maps seen below (J_1, K_1, J_0, K_0, Output from left to right).



Solving for the inputs:

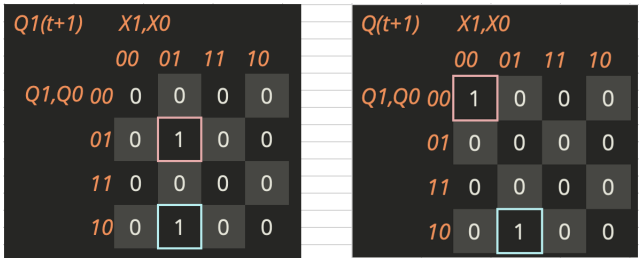
- $J_1 = Q_0\overline{X}_1X_0$
- $K_1 = \overline{X}_0 + X_1 + Q_0$
- $J_0 = \overline{Q}_1\overline{X}_1\overline{X}_0 + Q_1\overline{X}_1X_0$
- $K_0 = 1$
- $Output = Q_1Q_0X_1\overline{X}_0$
- all prime implicants are essential

The above inputs were then coded into a Verilog HDL dataflow module to model the state diagram (*figure 1*).

4.1.2 D flip-flop implementation

State Table - D Flip-flop						
Present state		**Inputs**		**Next state - Q = D**		**Outputs**
Q_1	Q_0	X_1	X_0	$Q_1(t+1)$	$Q_0(t+1)$	Correct
A ~ 00		0	0	0	1	0
		0	1	0	0	0
		1	0	0	0	0
		1	1	0	0	0
B ~ 01		0	0	0	0	0
		0	1	1	0	0
		1	0	0	0	0
		1	1	0	0	0
C ~ 10		0	0	0	0	0
		0	1	1	1	0
		1	0	0	0	0
		1	1	0	0	0
D ~ 11		0	0	0	0	0
		0	1	0	0	0
		1	0	0	0	1
		1	1	0	0	0

Similarly the same thing can be done with the D flip-flop implementation, using the property that D = Q(t+1) as shown in the third column of the state table. Since we are modelling the same state diagram, the final *output* would remain the same, i.e. the only thing we change is the D inputs. This can be determined from the K-maps below (D_1 on the left):



Solving for the inputs:

- $D_1 = \overline{Q}_1 Q_0 \overline{X}_1 X_0 + Q_1 \overline{Q}_0 \overline{X}_1 X_0$
- $D_0 = \overline{Q}_1 \overline{Q}_0 \overline{X}_1 \overline{X}_0 + Q_1 \overline{Q}_0 \overline{X}_1 X_0$

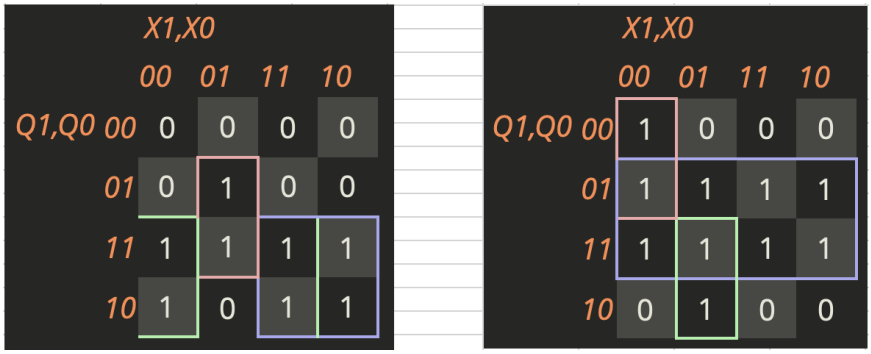
4.1.3 T flip-flop implementation

State Table - T Flip-flop							
Present state		**Inputs**		**Next state**		**Outputs**	**T (toggle)**	
Q_1	Q_0	X_1	X_0	$Q_1(t+1)$	$Q_0(t+1)$	Correct	T_1	T_0
A ~ 00		0	0	0	1	0	0	1
		0	1	0	0	0	0	0
		1	0	0	0	0	0	0
		1	1	0	0	0	0	0
B ~ 01		0	0	0	0	0	0	1
		0	1	1	0	0	1	1
		1	0	0	0	0	0	1
		1	1	0	0	0	0	1
C ~ 10		0	0	0	0	0	1	0
		0	1	1	1	0	0	1
		1	0	0	0	0	1	0
		1	1	0	0	0	1	0
D ~ 11		0	0	0	0	0	1	1
		0	1	0	0	0	1	1
		1	0	0	0	1	1	1
		1	1	0	0	0	1	1

This table is similar to how I implemented the JK flip-flop - through using an excitation table (shown below) and comparing the present and next states. Remember that we don't need another K-map for the output because the output is a function of the present state and the inputs (thus independent on the implementation chosen). $Q(n+1) = T\overline{Q} + \overline{T}Q$

Q_n	Q_{n+1}	T
0	0	0
0	1	1
1	0	1
1	1	0

Note that since I used 2 bits to represent 4 states, we will always need two flip flops to implement the state transition. The following K-maps have been provided (T_1 on the left, T_0 on the right).



Solving for the inputs:

- $T_1 = Q_0\overline{X}_1X_0 + Q_1\overline{X}_0 + Q_1X_1$
- $T_0 = \overline{Q}_1\overline{X}_1\overline{X}_0 + Q_0 + Q_1\overline{X}_1X_0$
- all prime implicants are essential

4.1.4 Assumptions

A couple of assumptions I made for the state diagram can be seen below.

- We can accept more than 4 digits as long as the last 4 digits are the correct digits - I made this assumption because some keypad locks in real life use a similar mechanism and it doesn't compromise the security at all.
- I also assumed that all the states are positive edge triggered i.e. it only transitions to the next state on the positive edge of the clock.
- Furthermore, the 'reset' input is defaulted as LOW and the positive edge of the input would reset the circuit, in other words it's an active-high reset.
- If you press three buttons and then enter, it would produce a wrong result.
- Our clear input is connected to **only** the FSM in *figure 1* and would directly clear the memory of the flip flops whereas our reset would reset ALL the flip flops in every FSM.
- Inputs will still be working during alarm state. This is so that you can get inside to turn the alarm off (which still requires the correct passcode).

4.2 Lock Timer

This subsection focus on the 'Lock Timer' part of the circuit which requires the lock to be LOW for 3 clock cycles after the correct passcode has been entered. The state diagram for this module can be seen above in *figure 2*. The state tables will be provided below.

State Table					
Present state		**Inputs**	**Next state**		**Outputs**
Q_1	Q_0	Correct	$Q_1(t+1)$	$Q_0(t+1)$	Lock
A ~ 00		0	0	0	1
		1	0	1	1
B ~ 01		X	1	0	0
		X	1	0	0
C ~ 10		X	1	1	0
		X	1	1	0
D ~ 11		X	0	0	0
		X	0	0	0

Since this is a linear state diagram with dependent states, logically the states wouldn't be able to be minimised. This is confirmed by the following implication table (no states are equal).

Implication table			
B	X		
C	X	X	
D	X	X	X
	A	B	C

Similarly, the three different types of flip-flops can be used to incorporate the circuit logic.

4.2.1 JK flip-flop implementation

State Table							
Present state		**Inputs**	**Next state**		**Outputs**	**J & K**	
Q_1	Q_0	Correct	$Q_1(t+1)$	$Q_0(t+1)$	Lock	J_1K_1	J_0K_0
A ~ 00		0	0	0	1	0X	0X
		1	0	1	1	0X	1X
B ~ 01		X	1	0	0	1X	X1
		X	1	0	0	1X	X1
C ~ 10		X	1	1	0	X0	1X
		X	1	1	0	X0	1X
D ~ 11		X	0	0	0	X1	X1
		X	0	0	0	X1	X1

This section implements the above circuit using a JK flip-flop. Let's delve into the K-maps and gate inputs (refer to the method used in 4.1.1 for more information):

Figure 4: J_1, K_1, J_0, K_0, Output from left to right

Solving for the inputs:
(we use X to represent the Correct input)

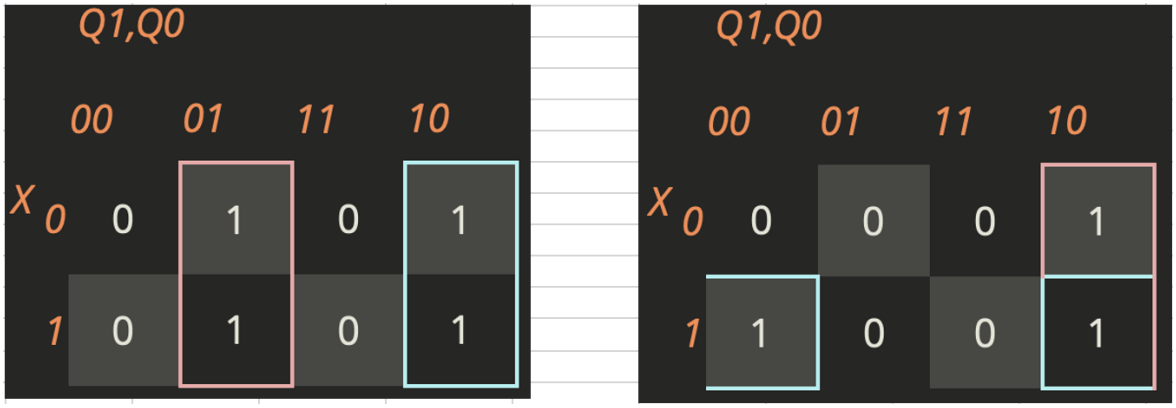
- $J_1 = Q_0$
- $K_1 = Q_0$
- $J_0 = Q_1 + X$
- $K_0 = \overline{Q}_1$
- $Lock = \overline{Q}_1\overline{Q}_0$
- all prime implicants are essential

Note that this is a Moore diagram so the output 'Lock' is not dependent on the input and is a function of the present state Q_1Q_0.

4.2.2 D flip-flop implementation

State Table					
Present state		**Inputs**	**Next state - D = Q**		**Outputs**
Q_1	Q_0	Correct	$Q_1(t+1)$	$Q_0(t+1)$	Lock
A ~ 00		0	0	0	1
		1	0	1	1
B ~ 01		X	1	0	0
		X	1	0	0
C ~ 10		X	1	1	0
		X	1	1	0
D ~ 11		X	0	0	0
		X	0	0	0

Recall that our D input is essentially the next state, thus we can do the same thing for the D flip-flop implementation as seen below (D_1 on the left):



Solving for the inputs:
(we use X to represent the Correct input)

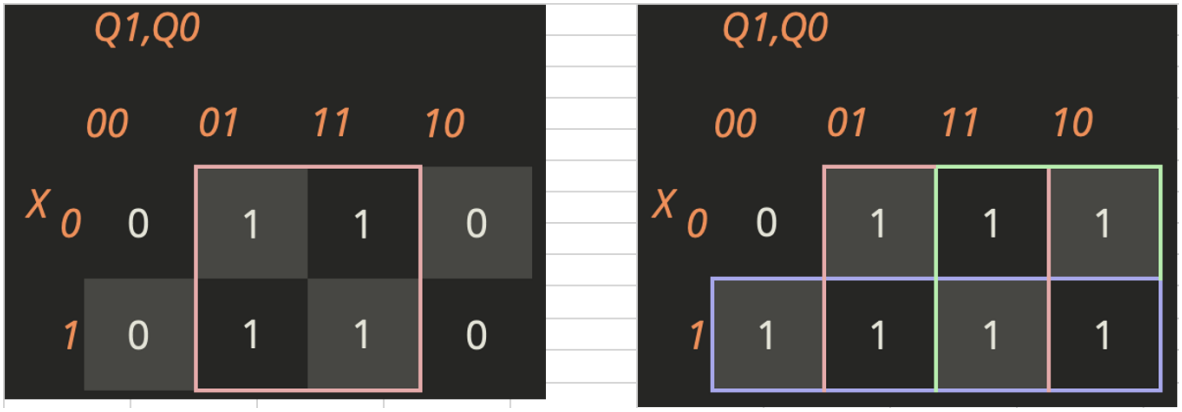
- $D_1 = \overline{Q}_1 Q_0 + Q_1 \overline{Q}_0$
- $D_0 = Q_1 \overline{Q}_0 + X \overline{Q}_0$
- all prime implicants are essential

The 'Lock' output remains the same.

4.2.3 T flip-flop implementation

State Table							
Present state		**Inputs**	**Next state**		**Outputs**	**T (toggle)**	
Q_1	Q_0	Correct	$Q_1(t+1)$	$Q_0(t+1)$	Lock	T_1	T_0
A ~ 00		0	0	0	1	0	0
		1	0	1	1	0	1
B ~ 01		X	1	0	0	1	1
		X	1	0	0	1	1
C ~ 10		X	1	1	0	0	1
		X	1	1	0	0	1
D ~ 11		X	0	0	0	1	1
		X	0	0	0	1	1

From comparing present and next states alongside analysis of the excitation table for the T flip-flop we can construct the two bit T_1T_0 inputs. K-maps are provided below (T_1 on the left).



Solving for the inputs:
(we use X to represent the Correct input)

- $T_1 = Q_0$
- $T_0 = Q_0 + Q_1 + X$
- all prime implicants are essential
- The 'Lock' output remains the same

4.3 Error counter

This subsection presents the implementation of the 'Error counter' module shown in *figure 3*. In order to effectively design the circuit for the above logic, I have chosen to use a four state FSM to cycle through the error counter, where the 'alarm' output would be HIGH once it reaches the final state after 3 consecutive errors. We also take the 2-bit input X_1X_0 where X_1 is equal to 'enter' and X_0 is equal to 'correct' which is an output from the Keypad inputs module.

Similar to the 'Lock timer' module (4.2) this FSM takes on a more linear structure because we are not able to go back to previous states without a reset. The following state table is used to describe this function.

State Table						
Present state		**Inputs**		**Next state**		**Outputs**
Q_1	Q_0	X_1	X_0	$Q_1(t+1)$	$Q_0(t+1)$	alarm
A ~ 00		0	0	0	0	0
		0	1	0	0	0
		1	0	0	1	0
		1	1	0	0	0
B ~ 01		0	0	0	1	0
		0	1	0	1	0
		1	0	1	0	0
		1	1	0	0	0
C ~ 10		0	0	1	0	0
		0	1	1	0	0
		1	0	1	1	0
		1	1	0	0	0
D ~ 11		X	X	1	1	1
		X	X	1	1	1
		X	X	1	1	1
		X	X	1	1	1

I used an implication table below to attempt to simplify the state machine and found that no states were equal.

Implication table			
B	X		
C	X	X	
D	X	X	X
	A	B	C

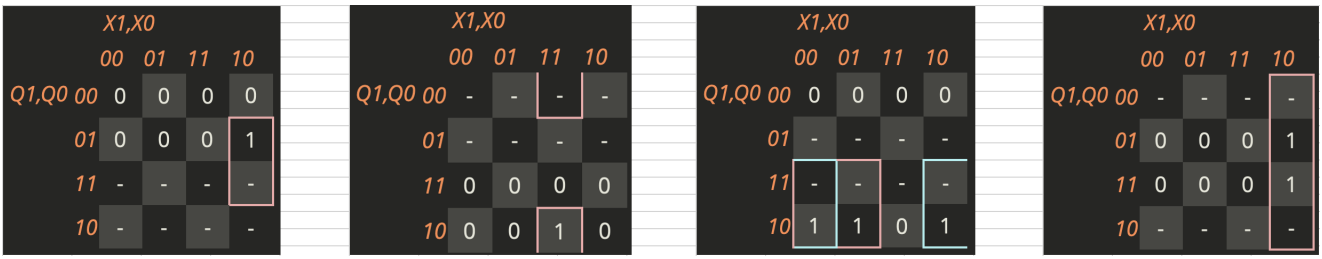
4.3.1 JK flip-flop implementation

State Table - JK flip-flop							
Present state		**Inputs**		**Next state**		**Outputs**	**J & K**	
Q_1	Q_0	X_1	X_0	$Q_1(t+1)$	$Q_0(t+1)$	alarm	J_1K_1	J_0K_0
A ~ 00		0	0	0	0	0	0X	0X
		0	1	0	0	0	0X	0X
		1	0	0	1	0	0X	0X
		1	1	0	0	0	0X	0X
B ~ 01		0	0	0	1	0	0X	X0
		0	1	0	1	0	0X	X0
		1	0	1	0	0	1X	X1
		1	1	0	0	0	0X	X1
C ~ 10		0	0	1	0	0	X0	1X
		0	1	1	0	0	X0	1X
		1	0	1	1	0	X0	1X
		1	1	0	0	0	X1	0X
D ~ 11		X	X	1	1	1	X0	X0
		X	X	1	1	1	X0	X0
		X	X	1	1	1	X0	X0
		X	X	1	1	1	X0	X0

The JK inputs were calculated by comparing the present state and next state while referring to an excitation table which I included again for convenience.

Q_n	Q_{n+1}	J	K
0	0	0	X
1	0	X	1
0	1	1	X
1	1	X	0

Without repeating myself too much, I will skip to the K-maps and input solutions (J_1, K_1, J_0, K_0 from left to right).



Solving for the inputs:

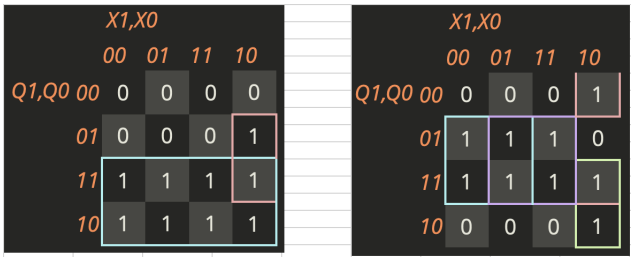
- $J_1 = Q_0 X_1 \overline{X}_0$
- $K_1 = \overline{Q}_0 X_1 X_0$
- $J_0 = Q_1 \overline{X}_1 + Q_1 \overline{X}_0$
- $K_0 = X_1 \overline{X}_0$
- $alarm = Q_1 Q_0$
- all prime implicants are essential

Note how I didn't actually include the K-map for alarm because we are working with a Moore state machine, and the output could be done by inspection since it's only HIGH when we are in the final state D.

4.3.2 D flip-flop implementation

State Table - D flip-flop						
Present state		**Inputs**		**Next state - D = Q**		**Outputs**
Q_1	Q_0	X_1	X_0	$Q_1(t+1)$	$Q_0(t+1)$	alarm
A ~ 00		0	0	0	0	0
		0	1	0	0	0
		1	0	0	1	0
		1	1	0	0	0
B ~ 01		0	0	0	1	0
		0	1	0	1	0
		1	0	1	0	0
		1	1	0	1	0
C ~ 10		0	0	1	0	0
		0	1	1	0	0
		1	0	1	1	0
		1	1	1	0	0
D ~ 11		X	X	1	1	1
		X	X	1	1	1
		X	X	1	1	1
		X	X	1	1	1

Again, our D_1D_0 input to our D flip-flop is equal to $Q_1(t+1)Q_0(t+1)$. I have included the following K-maps to convert this string of cases into Boolean logic (D_1 on the left).



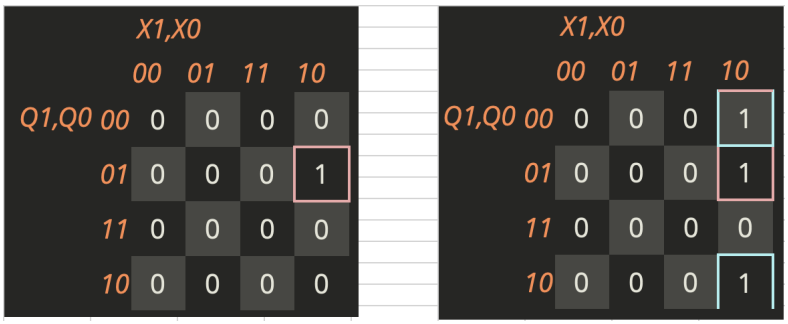
Solving for the inputs:

- $D_1 = Q_0X_1\overline{X}_0 + Q_1$
- D_1 prime implicants are all essential
- $D_0 = \overline{Q}_0X_1\overline{X}_0 + Q_0\overline{X}_1 + Q_0X_0 + Q_1X_1\overline{X}_0$
- The cyan and purple prime implicants are essential, the rest are just prime.

4.3.3 T flip-flop implementation

State Table - T flip-flop							
Present state		**Inputs**		**Next state**		**Outputs**	**T (toggle)**	
Q_1	Q_0	X_1	X_0	$Q_1(t+1)$	$Q_0(t+1)$	alarm	T_1	T_0
A ~ 00		0	0	0	0	0	0	0
		0	1	0	0	0	0	0
		1	0	0	1	0	0	1
		1	1	0	0	0	0	0
B ~ 01		0	0	0	1	0	0	0
		0	1	0	1	0	0	0
		1	0	1	0	0	1	1
		1	1	0	1	0	0	0
C ~ 10		0	0	1	0	0	0	0
		0	1	1	0	0	0	0
		1	0	1	1	0	0	1
		1	1	1	0	0	0	0
D ~ 11		0	0	1	1	1	0	0
		0	1	1	1	1	0	0
		1	0	1	1	1	0	0
		1	1	1	1	1	0	0

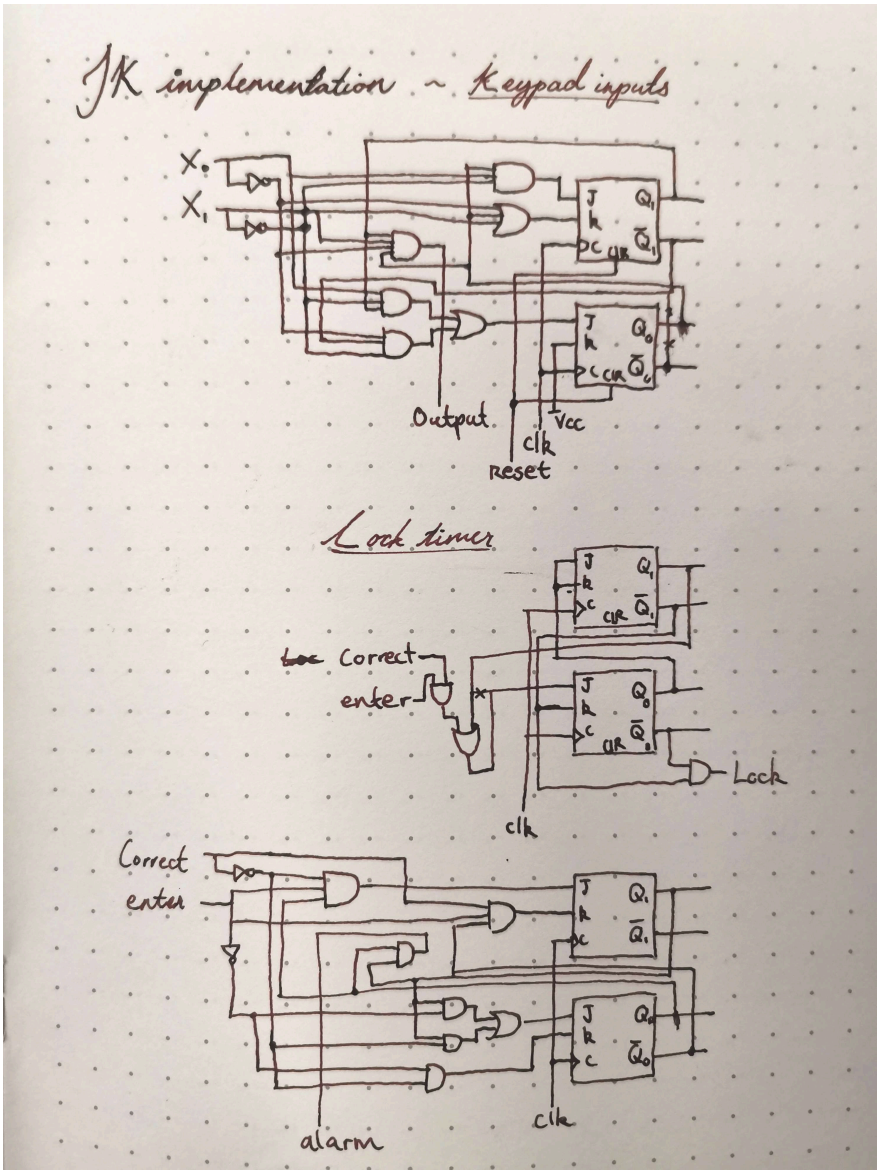
Here is a T flip-flop implementation of *figure 3* - the following K-maps can be seen below (T_1 on the left).



Solving for the inputs:

- $T_1 = \overline{Q}_1 Q_0 X_1 \overline{X}_0$
- $T_0 = \overline{Q}_1 X_1 \overline{X}_0 + \overline{Q}_0 X_1 \overline{X}_0$
- all prime implicants above are essential

4.4 Schematics & GIC



Above you can see the three logic diagrams that correspond to *figure 1, figure 2, figure 3* - Keypad inputs, Lock timer, Error counter respectively. I chose to use **JK flip-flops** for this implementation and the final calculated GIC is equal to 44, 30, and 42 which totals to 116. This is actually more efficient compared to a **D implementation** which I calculated to be 138 and **T** - 124.

5 Verilog HDL

5.1 Behavioural modelling

```
1  `timescale 1ns / 1ps
2
3  module ass2_behavioural(
4     input clk,
5     input reset,
6     input [11:0] keypad,
7     output reg lock,
8     output reg alarm
9      );
10
11    //--initialising the states--
12    reg [2:0] state, next_state;
13
14       //error count register
15    reg [1:0] error_count = 2'b00;
16
17
18       //These are the first four digits, plus the nth state for extra digits.
19    parameter
20    first = 3'b000,
21    second = 3'b001,
22    third = 3'b010,
23    fourth = 3'b011,
24
25
26    nth = 3'b100,
27    invalid = 3'b101;
28
29
30       //Now we can start writing the state transitions (refer to the template)
31       //Present state transition:
32    always@(posedge clk, posedge reset) begin
33       if (reset) begin
34          state <= first;
35          error_count = 0;
36          alarm <= 0;
37          lock <= 1;
38       end
39       else
40          state <= next_state;
41    end
```

```
44
45
46      always@*
47         if (error_count == 3) begin
48            alarm <= 1;
49            error_count <= 2'b00;
50         end
51
52
53         //Next state logic:
54      always@(keypad, posedge reset) begin
55         case(state)
56
57            3'b000   :  case(keypad)
58                           //assume pressing enter on first state does nothing
59                        12'b100000000000  :  next_state <= first;
60                        12'b010000000000  :  next_state <= first;
61                        12'b000000000010  :  next_state <= second;
62                        12'b000000000000  :  next_state <= first;
63                        default           :  next_state <= invalid;
64                     endcase
65
66
67            3'b001   :  case(keypad)
68                        12'b100000000000  :  next_state <= first;
69                        12'b010000000000  :  begin
70                                                next_state <= first;
71                                                error_count <= error_count + 1;
72                                             end
73                        12'b000000000100  :  next_state <= third;
74                        12'b000000000000  :  next_state <= second;
75                        default           :  next_state <= invalid;
76                     endcase
77
78            3'b010   :  case(keypad)
79                        12'b100000000000  :  next_state <= first;
80                        12'b010000000000  :  begin next_state <= first;
81                                             error_count <= error_count + 1;
82                                             end
83                        12'b000000000100  :  next_state <= fourth;
84                        12'b000000000000  :  next_state <= third;
85                        default           :  next_state <= invalid;
86                     endcase
```

```
87
88          3'b011   :  case(keypad)
89                        12'b100000000000  :  next_state <= first;
90                        12'b010000000000  :  begin
91                                               next_state <= first;
92                                               error_count <= error_count + 1;
93                                             end
94                        12'b000001000000  :  next_state <= nth;
95                        12'b000000000000  :  next_state <= fourth;
96                        default           :  next_state <= invalid;
97                     endcase
98
99          3'b100   :     case(keypad)
100                       12'b100000000000  :  next_state <= first;
101                       12'b010000000000  :  begin
102                                              next_state <= first;
103                                              error_count <= 2'b00;
104                                              lock <= 0;
105                                                 #150;
106                                              lock <= 1;
107                                            end
108                       12'b000000000000  :  next_state <= nth;
109                       default           :  next_state <= invalid;
110                    endcase
111
112         3'b101   :  case(keypad)
113                       12'b100000000000  :  next_state <= first;
114                       12'b010000000000  :  begin
115                                              next_state <= first;
116                                              error_count <= error_count + 1;
117                                            end
118                       default           :  next_state <= invalid;
119                    endcase
120      endcase
121   end
122endmodule
123
```

5.2 Dataflow JK implementation

I've also included my main structural file for this assignment as seen below.

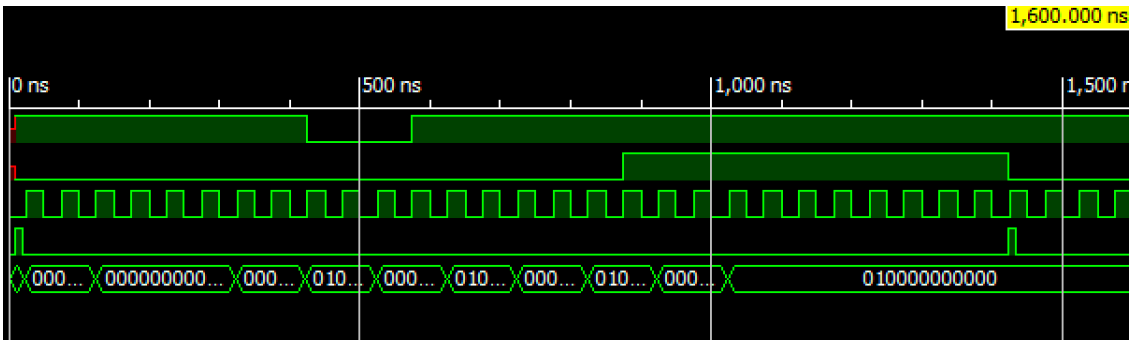
```
1  `timescale 1ns / 1ps
2
3  module ass2_structural(
4     input clk,
5     input reset,
6     input [11:0] keypad,
7     output lock,
8     output alarm
9      );
10
11    //this is simply our 2 bit input
12    wire [1:0] x;
13
14    //initially assigning to the keypad
15    assign x[1] = keypad[0]|keypad[3]|keypad[4]|keypad[5]|keypad[6]|keypad[7]|keypad[8]|keypa
16    assign x[0] = keypad[0]|keypad[2]|keypad[3]|keypad[4]|keypad[5]|keypad[7]|keypad[8]|keypa
17
18    //this is the flip flop implementation
19       //2 JK flip flops
20    wire [1:0] J = 0;
21    wire [1:0] K = 0;
22
23       //state register
24    reg [1:0] state = 0;
25
26    jk_ff jk0(J[0], K[0], reset, clk, state[0]);
27    jk_ff jk1(J[1], K[1], reset, clk, state[1]);
28
29    assign J[1] = state[0]&~x[1]&x[0];
30    assign K[1] = state[0]|~x[0]|x[1];
31    assign J[0] = (~state[1]&~x[1]&~x[0])|(state[1]&~x[1]&x[0]);
32    assign K[0] = 1;
33
34    assign lock = state[0];
35    assign alarm = state[1];
36
37 endmodule
```

6 Verifying using test benches

```
    `timescale 1ns / 1ps


module ass2_behavioural_tbw;

// Inputs
reg clk;
reg reset;
reg [11:0] keypad;

// Outputs
wire lock, alarm;

// Instantiate the Unit Under Test (UUT)
ass2_behavioural uut (
.clk(clk),
.reset(reset),
.keypad(keypad),
.lock(lock),
.alarm(alarm)
);

// Clock generation
   always #25 clk = ~clk; // Toggle clock every 20 time units

// Test scenario
initial begin
// reset the module
clk = 0;
reset = 0;
keypad = 0;
#10

reset = ~reset;
#10;
reset = ~reset;
#3;

keypad = 12'b000000000010;
#100;
keypad = 12'b000000000100;
```

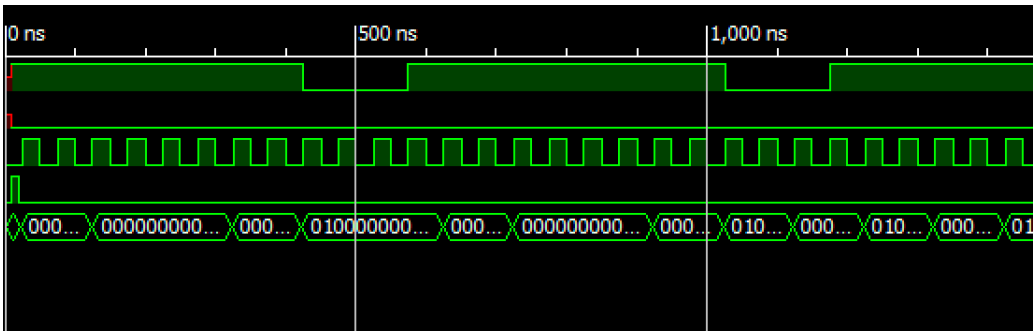
```
#100;
keypad = 12'b000000000100;
#100;
keypad = 12'b000001000000;
#100;
keypad = 12'b010000000000;
#100;

//testing
repeat (3) begin
keypad = 1;
#40;
keypad = 12'b010000000000;
#10;
end
#300;
reset = ~reset;
#10;
reset = ~reset;
#5;

end
endmodule
```

The test bench has been provided for the above code for both the behavioural and the structural. It goes through the basic cases such as alarm, reset-after-alarm, and unlock. We will test edge cases in the following test bench. Simulation results are provided in *figure 5*. Note that these test benches apply to both structural and behavioural.



6.1 Error cases

Two unlock signals in a row (testing the state cycle).

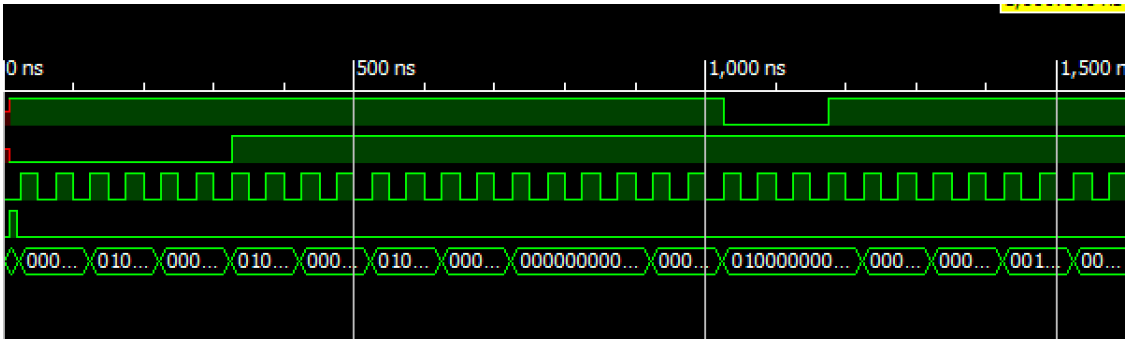



```
‘timescale 1ns / 1ps


module ass2_behavioural_tbw;

// Inputs
reg clk;
reg reset;
reg [11:0] keypad;

// Outputs
wire lock, alarm;

// Instantiate the Unit Under Test (UUT)
ass2_behavioural uut (
.clk(clk),
.reset(reset),
.keypad(keypad),
.lock(lock),
.alarm(alarm)
);

// Clock generation
   always #25 clk = ~clk; // Toggle clock every 20 time units

// Test scenario
initial begin
// reset the module
clk = 0;
reset = 0;
```

```
keypad = 0;
#10

reset = ~reset;
#10;
reset = ~reset;
#3;


//wrong input alarm
repeat (3) begin
keypad = 12'b000000000010;
#100;
keypad = 12'b010000000000;
#100;
end

keypad = 12'b000000000010;
#100;
keypad = 12'b000000000100;
#100;
keypad = 12'b000000000100;
#100;
keypad = 12'b000001000000;
#100;
keypad = 12'b010000000000;
#200;
keypad = 12'b000001000010;
#100;
keypad = 12'b000000000100;
#100;
keypad = 12'b001000000100;
#100;
keypad = 12'b000001000000;
#100;
keypad = 12'b010000001000;
#100;

//resetting the alarm
#300;
reset = ~reset;
#10;
reset = ~reset;
```

```
#5;

end
endmodule
```

The above test bench is testing inputs are still working during the alarm state. The simulation results are included below.



7 Conclusion

7.1 Conclusion

7.1.1 Conclusion

The end - you will not see any more simulations after this sentence.