

Solution to 11/28 example

State transition table (with assignments) ①

X	Present state			Next state			Output
	Q_2	Q_1	Q_0	Q_2^+	Q_1^+	Q_0^+	Z
	0	0	0	0	0	1	0
	0	0	1	0	1	0	0
	0	1	0	0	1	1	0
	0	1	1	0	1	1	0
	1	0	0	1	1	0	1
	1	0	1	1	1	0	1
	1	1	0	0	1	0	d
	1	1	1	d	d	d	d

J-K ff excitation table

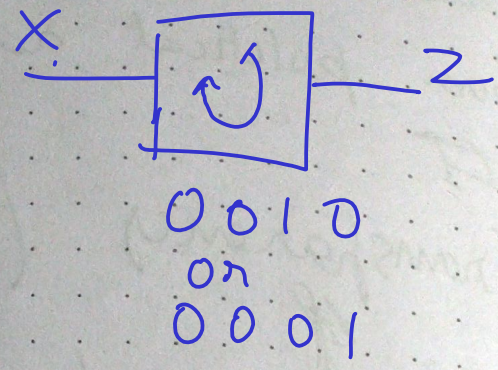
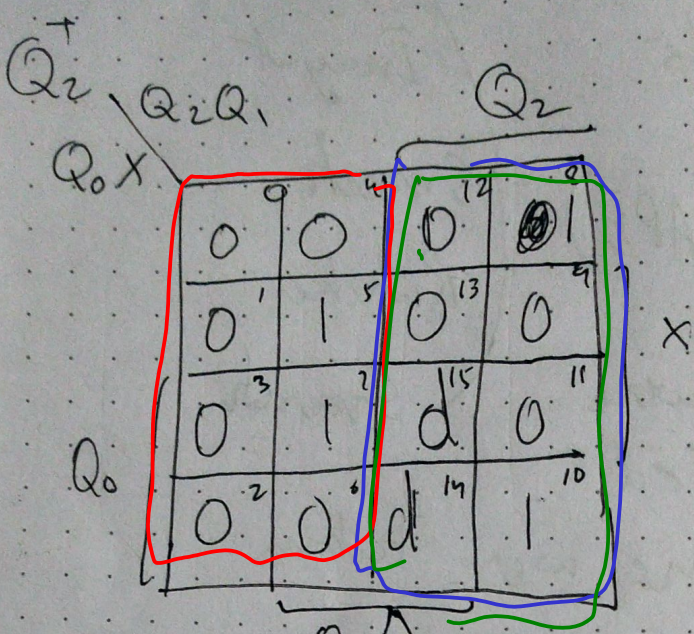
Q_2	Q_2^+	J_2	K_2
0	0	0	d
0	1	1	d
1	0	d	1
1	1	d	0

JK characteristic table

J	K	Q	Q^+
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	0

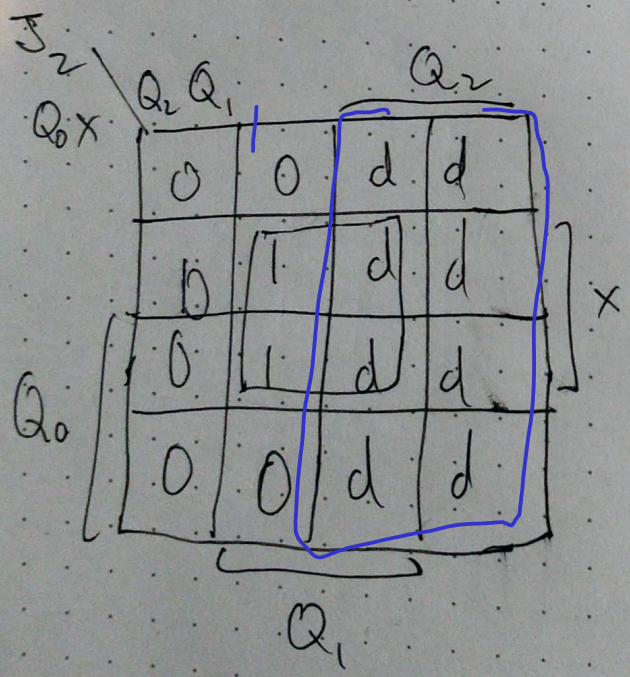
One way is draw Truth table for each $J_2 K_2$, $J_1 K_1$ and $J_0 K_0$

A shortcut is to draw K-map for Q_2^+ , Q_1^+ , Q_0^+ and then convert them to $J_2 K_2$, $J_1 K_1$, $J_0 K_0$ truth tables K-maps

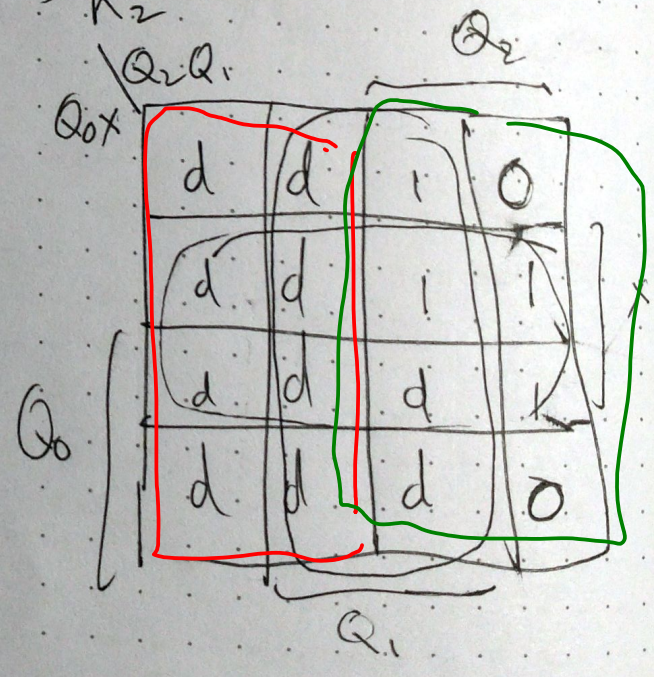


replace $Q_2=1$ with $J_2=d$

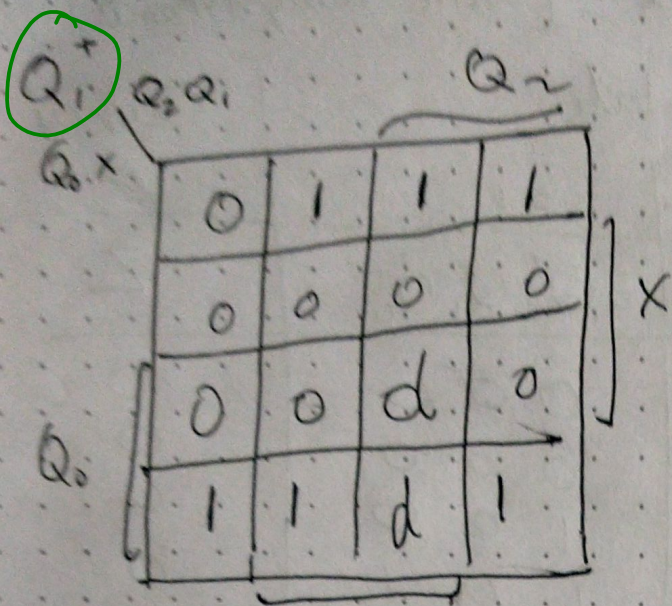
replace $Q_2=0$ with $K_2=d$ and flip Q_2^+



$J_2 = Q_1 \cdot X$



$K_2 = X + Q_1$

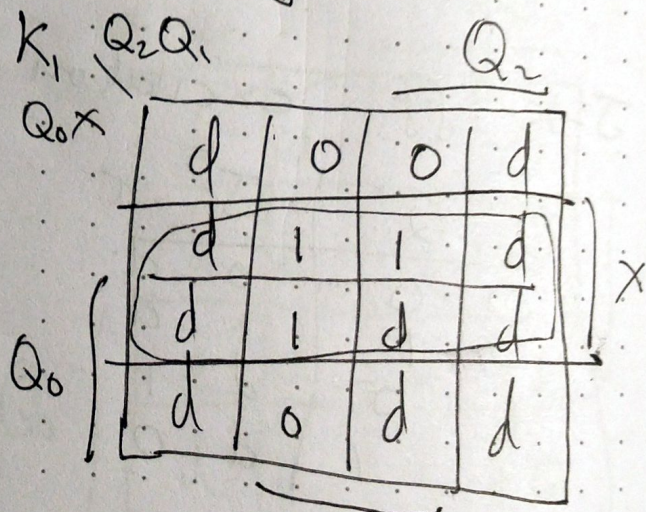
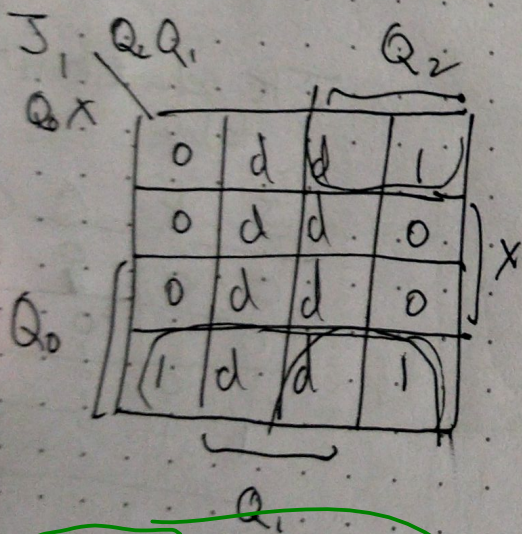


replace $Q_1=1$ with $J_1=d$

replace $Q_1=0$ with $K_1=d$ and flip rest

JK excitation table

Q	Q^*	J	K
0	0	0	d
0	1	d	0
1	0	d	d
1	1	d	0



$$J_1 = Q_0 \cdot \bar{X} + Q_2 \cdot \bar{X}$$

$$K_1 = X$$

Q_0 $Q_2 Q_1$

1	1	0	0
0	0	0	0
0	1	d	0
0	1	d	0

Q_0 Q_1

replace $Q_0=1$ with $J_0=d$

replace $Q_0=0$ with $K_0=d$ and flip rest.

J_0 $Q_2 Q_1$

1	1	0	0
0	0	0	0
d	d	d	d
d	d	d	d

J_0 Q_1

K_0 $Q_2 Q_1$

d	d	d	d
d	d	d	d
1	0	d	1
1	0	d	1

K_0 Q_1

$J_0 = \bar{Q}_2 \cdot \bar{Q}_1$

$K_0 = \bar{Q}_1$

Z $Q_2 Q_1$

0	0	1	0
0	0	d	1

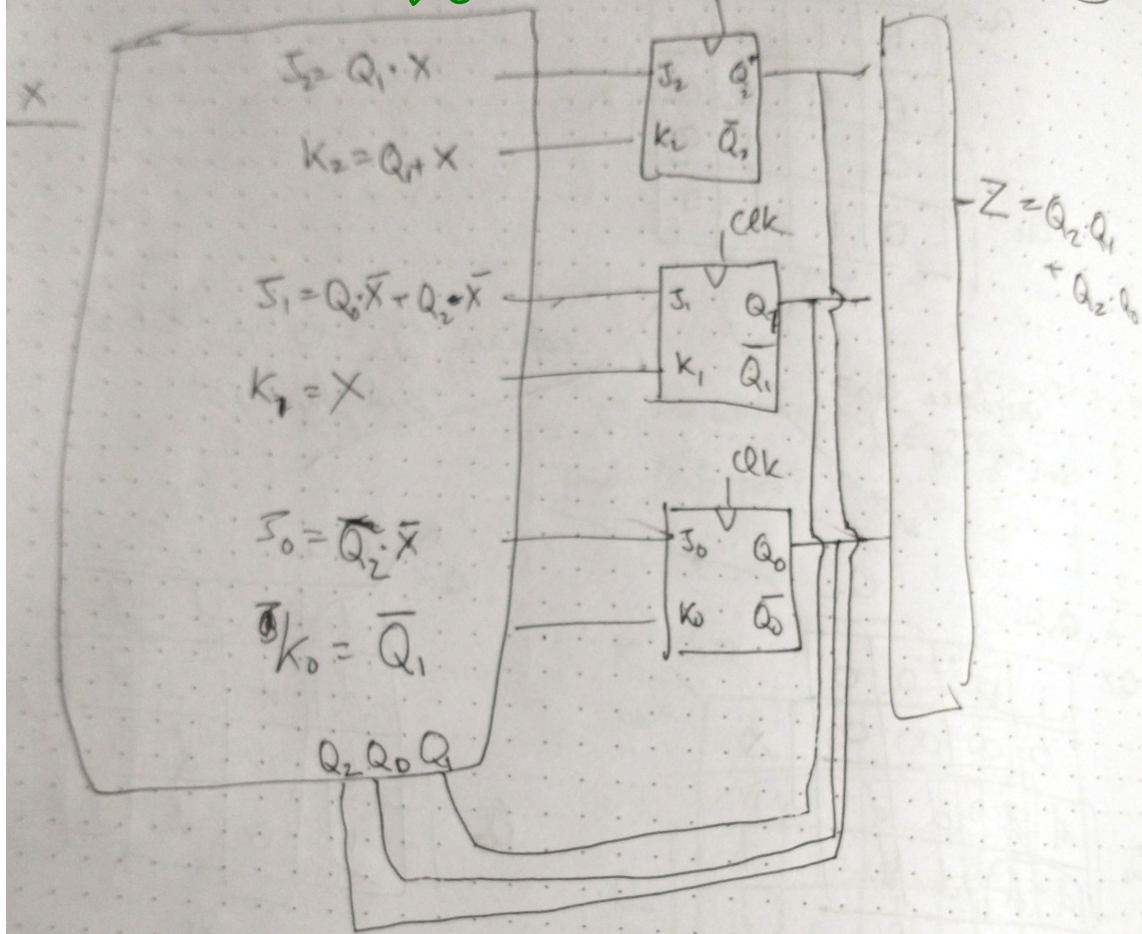
Z Q_1

$Z = Q_2 \cdot Q_1 + Q_2 \cdot Q_0$

Next state block

Reg

Output (5)



```

1  module seqdetector_top(
2  input clock,
3  input reset,
4  input X,
5  output reg Z
6  );
7
8  reg [2:0] state;
9  reg [2:0] next_state;
10
11 // Define states as constants
12 parameter [2:0]
13 S0 = 3'b000,
14 S1 = 3'b001,
15 S2 = 3'b010,
16 S3 = 3'b011,
17 S4 = 3'b100,
18 S5 = 3'b101,
19 S6 = 3'b110;
20
21
22 // Register Block
23 always_ff @(posedge clock or posedge reset) begin
24     if (reset)
25         state <= S0;
26     else
27         state <= next_state;
28 end
29
30 // Next state block
31 always_comb begin
32     // case statement is like if else but the condition is on a single variable
33     case (state)
34         S0:
35             next_state <= X ? S0 : S1;
36         S1:
37             next_state <= X ? S0 : S2;
38         S2:
39             next_state <= X ? S4 : S3;
40         S3:
41             next_state <= X ? S5 : S3;
42         S4:
43             next_state <= X ? S0 : S6;
44         S5:
45             next_state <= X ? S0 : S6;
46         S6:
47             next_state <= X ? S0 : S2;
48     endcase
49 end
50
51 // Output block (Moore)
52 always_comb begin
53     case (state)
54         S5: Z = 1'b1;
55         S6: Z = 1'b1;
56         default: Z = 1'b0;
57     endcase
58 end
59 endmodule
60

```

Registers
D-ff

System Verilog FAQs

Vikas Dhiman

November 30, 2022

Question 1. *Can you give us a template for all modules?*

There is no general template, but the following template will work for all **Synchronous Sequential** modules that do not call any other module.

```
1 // module keyword starts a module definition.
2 module module_named_foo(
3     // Every module should have a single bit clock and single bit reset signal
4     input wire [0:0] clock,
5     input wire [0:0] reset,
6     // All inputs to the module are declared as wires
7     input wire [bits1:0] input_1,
8     input wire [bits2:0] input_2,
9     ...
10    // All outputs from the module are declared as regs
11    output reg [bits3:0] output_1,
12    output reg [bits4:0] output_2
13);
14 // Every synchronous module will need some states
15 // States are always declared as registers
16 reg [bit5:0] state_1;
17 reg [bit6:0] state_2;
18 ...
19
20 // We have the choice of writing procedural code or structural code. Here we
21 // use procedural block. I will separate the procedural code into a
22 // register block and two combinational logic blocks
23
24 ////////////////////////////////////////////////////////////////////
25 // First block: Register block
26 ////////////////////////////////////////////////////////////////////
27 // Create some intermediate states
28 // These intermediate states could have been wires if we were using assign
29 // statement to create the combinational block. assign statement is easy to
30 // write only for very simple circuits like slowclock. For the rest, we use
31 // procedural code and reg for intermediate variables.
32 reg [bit5:0] next_state_1;
33 reg [bit6:0] next_state_2;
34 ...
35 // Always block that triggers only on the posedge of clock and posedge of
36 // reset signal.
37 // always_ff is same as always, but it ensures that a flip-flop circuit is
38 // synthesized.
39 always_ff @(posedge clock or posedge reset) begin
40     if (reset) begin
41         // This is the initialization block. You can assign initial values to your
42         // state here
43         state_1 <= 0;
```

```

44     state_2 <= 0;
45     ...
46     // Using the non-blocking assign '<=' in register block is recommended
47 end else begin
48     // At the rising edge next state is copied to current state
49     state_1 <= next_state_1;
50     state_2 <= next_state_2;
51     ...
52 end
53 end
54
55 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
56 // Second block: converts from current state and input to next state
57 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
58 // 1. Most of the logic of your state machine goes here
59 // 2. Note that combinational logic always block does not trigger on posedge
60 //    clock instead it triggers on any change in input.
61 // 3. You can also use always_comb instead of always @(*) which will ensure that
62 //    a combinational logic is synthesized.
63 // 4. Only next_state must be on the left hand side.
64 always @(*) begin
65     if (/*some conditions on states and inputs */) begin
66         next_state_1 = //some expression of states and inputs;
67         next_state_2 = //some expression of states and inputs;
68         ...
69         // Using the blocking assign '=' in combinational block is recommended
70     end else if (/*more conditions on states and inputs */) begin
71         next_state_1 = // some expression of states and inputs;
72         next_state_2 = // some expression of states and inputs;
73         ...
74     end else begin
75         next_state_1 = // some expression of state and inputs;
76         next_state_2 = // some expression of state and inputs;
77         ...
78     end
79 end
80
81 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
82 // Third block: converts from current state and input to output (Mealy)
83 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
84 always @(*) begin
85     if (/* condition on states and inputs */) begin
86         output_1 = // some expression of states and inputs
87         output_2 = // some expression of states and inputs
88         ...
89     end else if (/* condition on states and inputs */) begin
90         output_1 = // some expression of states and inputs
91         output_2 = // some expression of states and inputs
92         ...
93     end else begin
94         output_1 = // some expression of states and inputs
95         output_2 = // some expression of states and inputs
96         ...
97     end
98 end
99 endmodule

```

Question 2. *Can I combine the register block and the two combinational block into a single always block?*

Yes, you can. Not recommended, but it works. Most students are doing everything in a single always block. It does not mean that you should. Remember, you want to generate a circuit from this HDL code. It is helpful for your understanding to write HDL code that corresponds to circuit blocks. You should periodically check the RTL diagram in the Netlist viewer.

```

1 // module keyword starts a module definition.
2 module module_named_foo(
3     // Every module should have a single bit clock and single bit reset signal
4     input wire [0:0] clock,
5     input wire [0:0] reset,
6     // All inputs to the module are declared as wires
7     input wire [bits1:0] input_1,
8     input wire [bits2:0] input_2,
9     ...
10    // All outputs from the module are declared as regs
11    output reg [bits3:0] output_1,
12    output reg [bits4:0] output_2,
13);
14 // Every synchronous module will need some states
15 // States are always declared as registers
16 reg [bit5:0] state_1;
17 reg [bit6:0] state_2;
18 ...
19
20 // Always block that triggers only on the posedge of clock and posedge of
21 // reset signal.
22 // ‘always_ff’ is same as ‘always’, but it ensures that a flip-flop circuit is
23 // synthesized.
24 always_ff @(posedge clock or posedge reset) begin
25     if (reset) begin
26         // This is the initialization block. You can assign initial values to your
27         // state here
28         state_1 <= 0;
29         state_2 <= 0;
30         ...
31         // Using the non-blocking assign ‘<=’ in register block is recommended
32     end else if (/*some condition on states and inputs */)begin
33         // At the rising edge next state is copied to current state
34         state_1 <= /* some expression of states and inputs */;
35         state_2 <= /* some expression of states and inputs */;
36         ...
37         output_1 <= /* some expression of states and inputs */;
38         output_2 <= /* some expression of states and inputs */;
39         ...
40     end
41 end

```

Question 3. *How to connect multiple modules in the top level module?*

Please refer to Lab 7 for details of instantiating modules. There is confusion about whether reg can connect to wires or not. reg CAN connect to wires and vice versa.

```

1 module module_top(input wire CLOCK_50,
2                 input wire [2:0] BUTTON,
3                 ...);
4
5     // You can use wire and assign for simple combinational circuits.
6     // One a wire is assigned it cannot be assigned anything else.
7     wire reset;

```

```

8      assign reset = BUTTON[1];
9
10     // You can use wire to take the connect the output of one module to another.
11     wire CLOCK_10;
12     slowclock instance1_of_slowclock(CLOCK_50,
13     reset,
14     CLOCK_10);
15
16     // Here wire CLOCK_10 connects the output of slowclock to the input of
17     // foo
18     module_named_foo instance1_of_foo( CLOCK_10,
19     reset,
20     ...
21     ...);
22
23     endmodule

```

Question 4. *When to use register `reg` vs wire `wire`?*

Please refer back to Lab 6, when we learned about Verilog Procedural Operators. This is a quote from Lab 6 manual: “Another important aspect of the procedural always blocks is you would use registers on the left hand side of equations inside an always block. You would not use wires on the left hand side.” In general, the following rules can help:

1. Inputs of a module inside the module are `wire`. They are declared such even when the keyword `wire` is omitted.
2. Outputs of a module inside the module are `reg`. They are declared such even when `reg` is omitted.
3. Different modules are typically connected through a `wire`.
4. Only use `assign` with a `wire` on the left hand side. You CANNOT `assign` a `wire` more than one time.
5. When a symbol is on the left hand side of an equation inside the always block, it must be a `reg`.
6. `reg` are more general than `wire`. When in doubt use a `reg`.

Question 5. *When to use continuous assign `assign` vs non-blocking assign “`<=`” vs blocking assign “`=`”?*

The textbook has a very nice explanation of this usage in Section 4.5.4. I have reproduced the summary block here. In general, Chapter 4 is a useful read if you are still struggling with System Verilog programming.

Question 6. *What’s the deal with `initial` block?*

You should only use `initial` block for simulation. It is a non-synthesizable block, so it will not be converted into a circuit. Instead, use a reset signal and an `if (reset)` block to initialize your states.

SystemVerilog

1. Use `always_ff @(posedge clk)` and nonblocking assignments to model synchronous sequential logic.

```

always_ff @(posedge clk)
begin
    n1 <= d; // nonblocking
    q <= n1; // nonblocking
end

```

2. Use continuous assignments to model simple combinational logic.

```

assign y = s ? d1 : d0;

```

3. Use `always_comb` and blocking assignments to model more complicated combinational logic where the `always` statement is helpful.

```

always_comb
begin
    p = a ^ b; // blocking
    g = a & b; // blocking
    s = p ^ cin;
    cout = g | (p & cin);
end

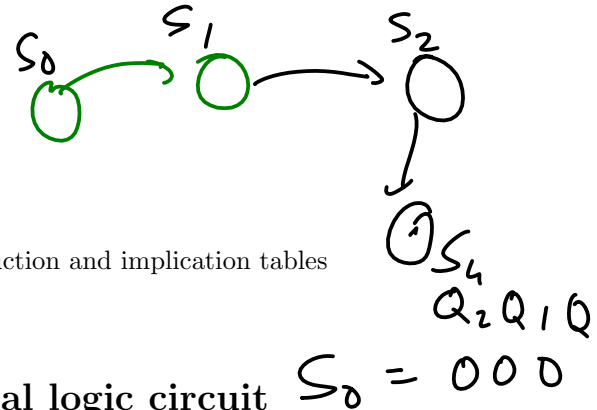
```

4. Do not make assignments to the same signal in more than one `always` statement or continuous assignment statement.

Sequential logic design

Vikas Dhiman for ECE275

November 30, 2022



1 Objectives

1. Perform a state assignment using the guideline method
2. Reduce the number of states in a state table using row reduction and implication tables
3. Partition a system into multiple state machines

2 Full procedure for designing sequential logic circuit

1. Convert the word problem to a state transition diagram. Let the states be $S_0, S_1, S_2, \dots, S_n$.
2. Draw state transition table with named states. For example,

Present State	Next State		Outputs	
	X = 0	X = 1	X=0	X=1
S_0	S_1	S_2	0	0
S_1	S_2	S_0	0	0
\vdots	\vdots	\vdots	\vdots	\vdots

3. State reduction step: Reduce the number of required states to a minimum. Eliminate unnecessary or duplicate states.
4. State assignment step: Assign each state a binary representation. For example,

State name	State assignments ($Q_2Q_1Q_0$)
S_0	000
S_1	001
\vdots	\vdots

5. Draw State assigned transition table. For example,

Inputs (X_1X_0)	Present State (Q_1Q_0)	Next State ($Q_1^+Q_0^+$)	Outputs (Z_1Z_0)
0 0	00	01	0 0
0 0	01	10	0 0
\vdots \vdots	\vdots	\vdots	\vdots \vdots

- (a) Use excitation tables to find truth tables for the combinational circuits. For example, the excitation table for J-K ff is

Q	Q ⁺	J	K
0	0	0	d
0	1	1	d
1	0	d	1
1	1	d	0

3 State assignment by guideline method [1, Section 8.2.5]

3.1 State Maps

Example 1. Draw a state map for a sequential assignment of the states

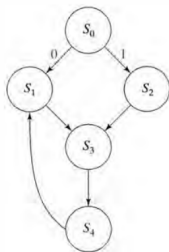
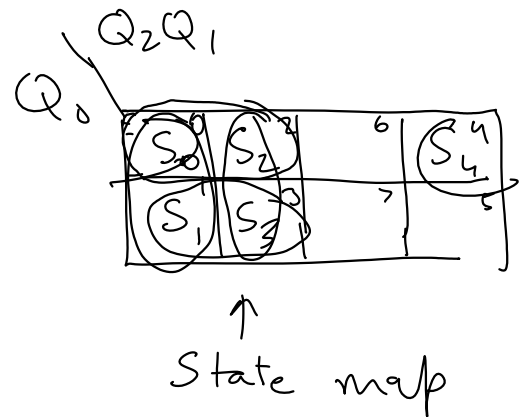


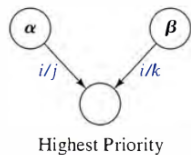
Figure 8.27 Five-state finite state machine.

State	Assignment		
	Q ₂	Q ₁	Q ₀
S ₀	0	0	0
S ₁	0	0	1
S ₂	0	1	0
S ₃	0	1	1
S ₄	1	0	0



3.2 Guideline method

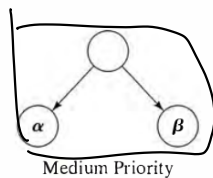
Guideline method states that the following states should be adjacent in the state map according to the following priorities:



Highest Priority

Highest priority: states with the same next state for a given input.

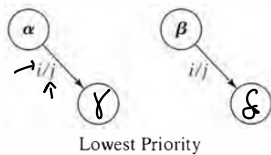
$$(\alpha, \beta) \quad (S_0, S_1)$$



Medium Priority

Medium priority: states that are next state of the same state.

$$(\alpha, \beta)$$



Lowest Priority

Lowest priority:

States with the same output corresponding to the same inputs.

$$(\alpha, \beta)$$

Figure 8.29 Adjacent assignment priorities.

Example 2. A state transition table is given. Find optimal state assignment by using the guideline method.

Reset should be assigned 00...0

Highest priority: (S_3, S_4)

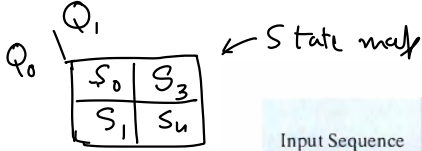
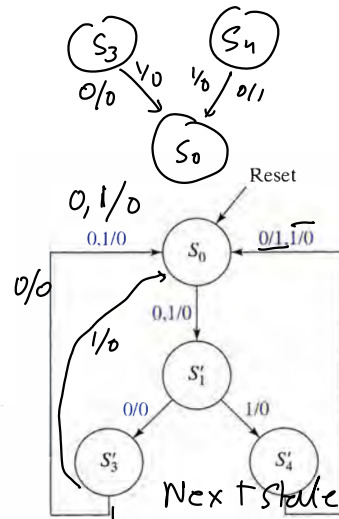
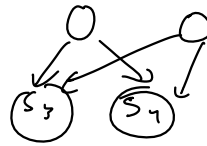
Medium priority pairs: $(S_3, S_4)^2$

Low priority groups:

(same output for same input)

0/0: (S_0, S_1, S_3)

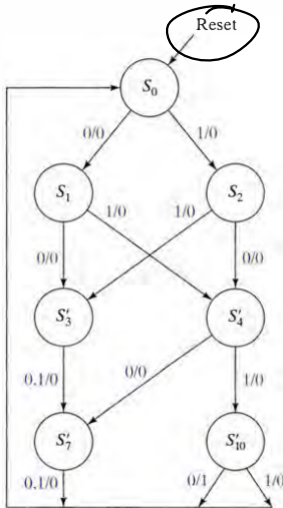
1/0: (S_0, S_1, S_3, S_4)



Input Sequence	Present State	Next State		Output	
		X=0	X=1	X=0	X=1
Reset	S_0	S_1	S_1	0	0
0 or 1	S_1	S_3	S_4	0	0
00 or 10	S_3	S_0	S_0	0	0
01 or 11	S_4	S_0	S_0	1	0

Figure 8.30 = Reduced State diagram for 3-bit sequence detector.

Example 3. Draw a Mealy FSM for detecting binary string 0110 or 1010. The machine returns to the reset state after each and every 4-bit sequence. Draw the state transition diagram on your own as practice problem. The state transition diagram is given here. Find optimal state assignment by using the guideline method.



Highest priority: $(S_7, S_{10}), (S_3, S_4)$

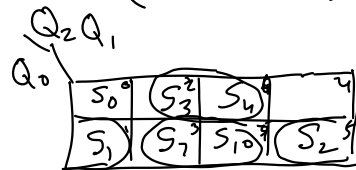
Medium priority: $(S_1, S_2), (S_3, S_4)^2, (S_7, S_{10})$

Lowest priority: 0/0: $(S_0, S_1, S_2, S_3, S_4, S_7)$

1/0: $(S_0, S_1, S_2, S_3, S_4, S_7, S_{10})$

of states = 7

of bits = 3



4 State reduction by implication chart

Example 4. Design a Mealy FSM for detecting binary sequence 010 or 0110. The machine returns to reset state after each and every 3-bit sequence. For now the state transition table is given. Reduce the following state transition table

	Q_2	Q_1	Q_0
S_0	0	0	0
S_1	0	0	1
S_2	1	0	1
S_3	0	1	0
S_4	1	1	0
S_7	0	1	1
S_{10}	1	1	1

Input Sequence	Present State	Next State		Output	
		$X=0$	$X=1$	$X=0$	$X=1$
Reset	S_0	S_1	S_2	0	0
0	S_1	S_3	S_4	0	0
1	S_2	S_5	S_6	0	0
00	S_3	S_0	S_0	0	0
01	S_4	S_0	S_0	1	0
10	S_5	S_0	S_0	0	0
11	S_6	S_0	S_0	1	0

4.1 Implication chart Summary

The algorithms for state reduction using the implication chart method consists of the following steps

1. Construct the implication chart, consisting of one square for each possible combination of states taken two at a time.
2. For each square labeled by states S_i and S_j , if the outputs of the states differ, mark the square with an X ; the states are not equivalent. Otherwise, they may be equivalent. Within the square write implied pairs of equivalent next states for all input combinations.
3. Systematically advance through the squares of the implication chart. If the square labeled by states S_i, S_j contains an implied pair S_m, S_n and square S_m, S_n is marked with an X , then mark S_i, S_j with an X . Since S_m, S_n are not equivalent, neither are S_i, S_j .
4. Continue executing Step 3 until no new squares are marked with an X .
5. For each remaining unmarked square S_i, S_j , we can conclude that S_i, S_j are equivalent.

References

- [1] Randy Katz and Gaetano Barriello. *Contemporary Logic Design*. Prentice Hall, 2004.

Ask for reference PDF if you need it.